Titre: Title:	Détection d'ensembles irréductibles incohérents dans des problèmes de satisfaction de contraintes irréalisables
Auteur: Author:	Christian Desrosiers
Date:	2004
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Desrosiers, C. (2004). Détection d'ensembles irréductibles incohérents dans des problèmes de satisfaction de contraintes irréalisables [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8176/

UNIVERSITÉ DE MONTRÉAL

DÉTECTION D'ENSEMBLES IRRÉDUCTIBLES INCOHÉRENTS DANS DES PROBLÈMES DE SATISFACTION DE CONTRAINTES IRRÉALISABLES

CHRISTIAN DESROSIERS
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AVRIL 2004

Your file Votre référence
ISBN: 0-612-91936-6
Our file Notre référence
ISBN: 0-612-91936-6

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

DÉTECTION D'ENSEMBLES IRRÉDUCTIBLES INCOHÉRENTS DANS DES PROBLÈMES DE SATISFACTION DE CONTRAINTES IRRÉALISABLES

présenté par: DESROSIERS Christian

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. GAGNON Michel, Ph.D., président

M. GALINIER Philippe, Doct., membre et directeur de recherche

M. HERTZ Alain, Doct. ès Sc., membre et codirecteur de recherche

M. PESANT Gilles, Ph.D., membre

REMERCIEMENTS

Je tiens tout d'abord à exprimer ma reconnaissance envers mon directeur de recherche, M. Philippe Galinier, et mon codirecteur de recherche, M. Alain Hertz, pour les conseils précieux qu'ils m'ont donnés, pour le soutien financier qu'ils m'ont accordé, ainsi que pour le temps qu'ils ont consacré à diriger ce mémoire.

Je remercie également les professeurs Michel Gagnon et Gilles Pesant d'avoir accepté d'examiner ce mémoire dans de si courts délais.

Je remercie enfin ma conjointe, ma famille et mes amis qui m'ont tous inspiré et encouragé durant l'élaboration de ce mémoire.

RÉSUMÉ

La satisfaction de contraintes est un paradigme qui permet de modéliser et résoudre une variété de problèmes d'optimisation combinatoire réels provenant de multiples disciplines. Un problème de satisfaction de contraintes (CSP) consiste à affecter à chaque variable x d'un ensemble $\mathcal{X}$ une valeur de son domaine D_x de telle sorte que toutes les contraintes d'un ensemble $\mathcal{C}$ soient satisfaites. Un CSP est dit réalisable si une telle affectation existe. Un sous-ensemble de contraintes est dit incohérent s'il n'existe aucune affectation satisfaisant toutes les contraintes de ce sous-ensemble. Un ensemble incohérent irréductible (IIS) de contraintes est un ensemble incohérent de contraintes qui devient cohérent lorsque n'importe laquelle de ses contraintes est retirée. De même, un sous-ensemble de variables est incohérent s'il n'existe aucune affectation satisfaisant toutes les contraintes n'impliquant que des variables de ce sous-ensemble. Un ensemble incohérent irréductible (IIS) de variables est donc un ensemble incohérent de variables qui devient cohérent lorsque n'importe laquelle de ses variables est retirée.

Ce mémoire présente des algorithmes pour obtenir des IIS de contraintes et de variables dans un CSP incohérent. Ces algorithmes ont été testés sur des instances connues et générées aléatoirement du problème de k-coloriage de graphe, qui consiste à déterminer s'il existe une coloration des sommets du graphe utilisant au plus k couleurs et telle que deux sommets adjacents soient de couleurs différentes. La plus petite valeur de k pour laquelle une telle coloration existe est le nombre chromatique du graphe. Les expériences présentées dans ce mémoire avaient comme buts d'évaluer les algorithmes de détection d'IIS, ainsi que l'utilité des IIS pour démontrer l'incohérence d'un CSP. Les résultats ont permis d'améliorer la borne inférieure sur le nombre chromatique de certains graphes connus, et même de fixer le nombre chromatique de graphes pour lesquels cette valeur n'était pas connue.

ABSTRACT

Constraint satisfaction is a paradigm that allows to model and solve a great variety of real-life optimization problems from many disciplines. A constraint satisfaction problem (CSP) consists in assigning to each variable x of a set $\mathcal{X}$ a value of its domain D_x such that all the constraints of a set $\mathcal{C}$ are satisfied. A CSP is solvable if such an assignment exists. A subset of constraints is said to be inconsistent if there is no assignment satisfying all the constraints of this subset. An irreducible inconsistent set (IIS) of constraints is an inconsistent set of constraints that becomes consistent when any of its constraints is removed. Similarly, a subset of variables is inconsistent if there is no assignment that satisfies all the constraints involving only variables of this subset. An irreducible inconsistent set (IIS) of variables is thus an inconsistent set of variables that becomes consistent when any of its variables is removed.

This thesis presents algorithms to find constraint and variable IIS in an inconsistent CSP. These algorithms were tested on known and randomly generated instances of the graph k-coloring problem. This problem consists in finding a coloring of the vertices of a graph using at most k colors and such that no two vertices linked by an edge have the same color. The smallest value of k for which such a coloring exists is the chromatic number of the graph. The experiments presented in this thesis aim at evaluating the IIS detection algorithms, as well as the usefulness of IISs in proving that a given CSP is inconsistent. Results of these experiments have enabled to improve the lower bound on the chromatic number of certain graphs, and even set the chromatic number of graphs for which this value was not known.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES NOTATIONS ET DES SYMBOLES

CN : Réseau de contraintes (*Constraint network*)

CSP : Problème de satisfaction de contraintes (*Constraint Satisfaction Problem*)

HS : Hitting set (*Hitting Set*)

IIS : Ensemble incohérent irréductible (*Irreducible Inconsistent Set*)

LP : Programme linéaire (Linear Program)

LSCP : Problème de recouvrement d'ensembles de grande taille (*Large Set Covering Problem*)

MCH : Heuristique de Moindre-conflit (*Min-Conflict Heuristic*)

MCSP : Problème de satisfaction maximale de contraintes (*Maximum Constraint Satisfaction Problem*)

MHS : Hitting set de cardinalité minimum (*Minimum Cardinality Hitting Set*)

MLSCP : Problème de recouvrement minimum d'ensembles de grande taille (*Minimum Large Set Covering Problem*)

MPCSP : Problème de satisfaction partielle maximum de contraintes (*Maximum Partial Constraint Satisfaction Problem*)

MPWCSP : Problème de satisfaction partielle maximale de contraintes pondérées (*Maximum Partial Weighted Constraint Satisfaction Problem*)

MWCSP : Problème de satisfaction maximale de contraintes pondérées (*Maximum Weighted Constraint Satisfaction Problem*)

MWFS : Sous-système réalisable de poids maximum (*Maximum Weight Feasible Subsystem*)

MWHS : Hitting set de poids minimum (*Minimum Weight Hitting Set*)

MWIC : Hitting set d'IIS de poids minimum (*Minimum Weight IIS Cover*)

PCSP : Problème de satisfaction partielle de contraintes (*Partial Constraint Satisfaction Problem*)

SAT : Problème de satisfaction booléenne

LISTE DES TABLEAUX

LISTE DES ANNEXES

CHAPITRE 1

INTRODUCTION

1.1 Satisfaction de contraintes

La satisfaction de contraintes (CS) (Mackworth, 1987; Tsang, 1993) est un moyen efficace de modéliser et de résoudre une grande variété de problèmes, telle que la confection d'horaires (Fox et Sadeh-Koniepcol, 1990), la configuration de produits (Mittal et Falkenhainer, 1990), ainsi que la planification et l'allocation de ressources (Choueiry, 1994). Chacun de ces problèmes peut être représenté comme un ensemble de variables auxquelles il faut assigner une valeur, en respectant un certain ensemble de contraintes spécifiques. La tâche peut être de montrer qu'il existe une solution, de trouver une solution, de trouver toutes les solutions, ou, finalement, de trouver une solution optimale selon un critère donné, pouvant être sous la forme d'une fonction objectif. La sous-section suivante offre une définition formelle de la satisfaction de contraintes.

1.1.1 Définition formelle

Définition Un réseau de contraintes (CN) est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, où $\mathcal{X} = \{x_1, \ldots, x_{|\mathcal{X}|}\}$ est un ensemble fini de variables, $\mathcal{D} = \{D_{x_1}, \ldots, D_{x_{|\mathcal{X}|}}\}$ est un ensemble tel que D_x est le domaine fini de la variable $x \in \mathcal{X}$, et $\mathcal{C} = \{c_1, \ldots, c_{|\mathcal{C}|}\}$ est un ensemble fini de contraintes sur les variables de $\mathcal{X}$.

Définition Étant donné un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, une affectation (instanciation) a de $Y \subseteq$

$\mathcal{X}$ est une fonction qui associe à chaque variable $y \in Y$ une valeur $a(y) \in D_y$. Une affectation est dite complète si elle assigne une valeur à chaque variable de $\mathcal{X}$ (i.e. $Y = \mathcal{X}$), sinon elle est partielle. Soit un ensemble de variables $Z \subseteq Y$, la projection $\pi(a, Z)$ de a sur Z est une affectation a' de Z telle que $a'(z) = a(z)\ \forall z \in Z$.

Définition Étant donné un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, la contrainte $c \in \mathcal{C}$ agissant sur l'ensemble de variables $\mathcal{X}(c)$ est une fonction:

$$c : \prod_{x_i \in \mathcal{X}(c)} D_i \rightarrow \{0, 1\}$$

qui, pour chaque affectation a possible de $\mathcal{X}(c)$, associe la valeur 0 si c est satisfaite par a, ou la valeur 1 si c est violée par a. On note $\mathcal{R}(c)$, l'ensemble des affectations a de $\mathcal{X}(c)$ telles que $c(a) = 0$. On note, par ailleurs, $\mathcal{C}(x)$ l'ensemble de contraintes agissant sur une variable $x \in \mathcal{X}$.

Définition Étant donné un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ et une affectation a de $Y \subseteq \mathcal{X}$, la contrainte $c \in \mathcal{C}$ est satisfaite par a si:

$$c\big(\pi\big(a, \mathcal{X}(c)\big)\big) = 0$$

Dans le cas où une ou plusieurs variables de $\mathcal{X}(c)$ ne sont pas instanciées (i.e. $\mathcal{X}(c) \nsubseteq Y$), on dit que c est satisfaite s'il est possible de compléter l'affectation pour l'ensemble de variables $\mathcal{X}(c) \setminus Y$ de telle que sorte que c soit satisfaite. L'affectation a est dite légale si elle satisfait toutes les contraintes de $\mathcal{C}$:

$$\sum_{c \in \mathcal{C}} c\big(\pi\big(a, \mathcal{X}(c)\big)\big) = 0$$

Définition Étant donné un CN défini par $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, le problème de satisfaction de contraintes (CSP) consiste à trouver une affectation complète légale de $\mathcal{X}$ ou à

démontrer qu'il n'en existe aucune. Une instance de CSP est dite soluble, cohérente, ou réalisable si elle possède au moins une solution, sinon elle est insoluble, incohérente ou irréalisable. Soit P un CSP donné, on note $\mathcal{S}$ l'ensemble des solutions de P.

La définition classique du CSP est de déterminer si une affectation complète légale existe. Il y a, cependant, plusieurs variantes de ce problème. Ainsi, dans le cas de CSP solubles, il peut être demandé de trouver une solution optimale selon une certaine fonction objectif. Par ailleurs, pour les CSP insolubles, il peut être exigé de déterminer une affectation complète satisfaisant un nombre maximum de contraintes. Ce dernier problème, connu sous le nom de MCSP (ou MaxCSP) (Freuder et Wallace, 1992), est défini comme suit:

Définition Étant donné un CN (X, D, C), le problème de satisfaction maximum de contraintes (MCSP) consiste à trouver une affectation complète a telle que le nombre de contraintes satisfaites est maximum. Une définition équivalente est de trouver une affectation complète a minimisant la fonction objectif f suivante:

$$f(a) = \sum_{c \in C} c\big(\pi\big(a, \mathcal{X}(c)\big)\big)$$

Le cadre du CSP classique ne permet de représenter que des problèmes de satisfaction. Dans cette optique, les contraintes du CN ont toutes le même poids et peuvent uniquement être complètement satisfaites ou complètement violées. Cependant, dans la plupart des problèmes réels, les contraintes n'ont pas toutes la même importance dans le CN, et ont un degré de satisfaction pouvant prendre plusieurs valeurs. Les réseaux de contraintes pondérées (de Givry et al., 2003) sont une extension des CN classiques, où chaque contrainte possède un coût correspondant à son degré de satisfaction:

Définition Un réseau de contraintes pondérées est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, où $\mathcal{X}$ est un ensemble fini de variables, $\mathcal{D}$ est un ensemble tel que $D_x \in \mathcal{D}$ est le domaine fini de la variable $x \in \mathcal{X}$, et $\mathcal{C}$ est un ensemble fini de contraintes sur les variables de $\mathcal{X}$. Une contrainte pondérée $c \in \mathcal{C}$, agissant sur l'ensemble de variables $\mathcal{X}(c)$, est une fonction:

$$c : \prod_{x \in \mathcal{X}(c)} D_x \rightarrow \mathbb{R}^+$$

qui, pour chaque affectation possible a de $\mathcal{X}(c)$, associe une valeur réelle $c(a)$ correspondant au degré d'insatisfaction de c. Ainsi, $c(a) = 0$ si c est complètement satisfaite par a, et $c(a) > 0$ si c est violée par a.

La notion de coût d'une contrainte c peut être illustrée à l'aide de la contrainte "toutes différentes" (*all-different*) qui impose aux variables de $\mathcal{X}(c)$ d'avoir toutes des valeurs différentes dans une affectation a de $\mathcal{X}(c)$. Si toutes les variables de $\mathcal{X}(c)$ ont des valeurs différentes dans a, la contrainte est complètement satisfaite. Si, par contre, une ou plusieurs variables ont la même valeur, le coût $c(a)$ de cette contrainte est, par exemple, donné par le nombre de variables ayant la même valeur qu'une autre variable dans l'ensemble. Une autre définition possible de $c(a)$ est $|\mathcal{X}(c)|$ moins le nombre de valeurs différentes affectées aux variables de cet ensemble. En somme, il existe plusieurs façons de modéliser le degré de satisfaction d'une contrainte.

Comme le MCSP pour les CN classiques, les CN pondérés ont également un problème associé de satisfaction maximum:

Définition Étant donné un CN pondéré (X, D, C), le problème de satisfaction maximum de contraintes (MWCSP) consiste à trouver une affectation complète a

minimisant la fonction objectif f suivante:

$$f(a) = \sum_{c \in \mathcal{C}} c\big(\pi\big(a, \mathcal{X}(c)\big)\big)$$

On remarque que le MWCSP ne diffère du MCSP que par la définition des contraintes, et que la fonction objectif est la même pour ces deux problèmes. En fait, le MWCSP est une généralisation du MCSP puisque n'importe quelle instance de MCSP peut être traduite en MWCSP simplement en définissant toute contrainte $c \in \mathcal{C}$ comme une fonction retournant une valeur réelle valant toujours 0 si c est satisfaite, et toujours 1 si c est violée.

Il existe, finalement, un problème de satisfaction général pour les CSP insolubles, appelé le problème de satisfaction partielle de contraintes (PCSP) (Freuder et Wallace, 1992). Le PCSP permet d'optimiser la satisfaction d'un CSP sur-contraint en affaiblissant certaines de ses propriétés. En ce sens, le PCSP consiste à trouver une solution pour un nouveau problème créé en relaxant le CSP original, telle que la relaxation est minimale selon un critère donné.

Définition Soit un quintuplet $(\mathcal{H}, \leq, d, P, \beta)$, où $\mathcal{H}$ est un ensemble de CSP, $\leq$ un ordre partiel sur les CSP de $\mathcal{H}$ tel que $P_i \leq P_j$ si et seulement si $\mathcal{S}(P_i) \supseteq \mathcal{S}(P_j)$, $d : \mathcal{H}^2 \rightarrow \mathbb{R}^+$ une fonction telle que $d(P_i, P_j)$ est la distance entre P_i et P_j, $P \in \mathcal{H}$ un CSP insoluble donné, et $\beta \in \mathbb{R}^+$ une borne nécessaire sur la distance entre P et n'importe quel CSP $P' \in \mathcal{H}$. Le problème de satisfaction partielle de contraintes (PCSP) consiste à trouver un CSP soluble $P' \in \mathcal{H}$ tel que:

$$d(P, P') < \beta$$

Le problème de satisfaction partielle maximum des contraintes (MPCSP) consiste

à trouver une solution P' au PCSP telle que $d(P, P')$ est minimum.

La fonction de distance d entre deux CSP P_i et P_j peut être définie de plusieurs façons. Une formulation possible est:

$$d = |(\mathcal{S}(P_i) \cup \mathcal{S}(P_j)) \setminus (\mathcal{S}(P_i) \cap \mathcal{S}(P_j))|$$

soit le nombre de solutions n'étant pas partagées entre $\mathcal{P}_i$ et $\mathcal{P}_j$. Cette formulation mesure le nombre de solutions rajoutées à P' par la relaxation. D'autres définitions plus simples sont, par exemple, la différence du nombre de contraintes (i.e. $|\mathcal{C}|$) ou du nombre de variables (i.e. $|\mathcal{X}|$) entre P_i et P_j. On remarque, par ailleurs, que d respecte l'ordre $\leq$ imposé sur $\mathcal{H}$. Comme P est sur-contraint il ne possède aucune solution (i.e. $\mathcal{S}(P) = \emptyset$), et tout CSP $P' \in \mathcal{H}$ donne un ordre $P' \leq P$ puisque $\mathcal{S}(P') \supseteq \mathcal{S}(P)$. Dans le cas où P' est lui aussi insoluble, on a $d(P, P') = 0$, car ces deux CSP ont le même nombre de solutions, soit 0. Par ailleurs, si P' est soluble, on a $P' < P$ et $d(P, P') > 0$. Ainsi, plus on relaxe P, plus le CSP relaxé P' possède un nombre important de solutions, et conséquemment, plus la distance $d(P, P')$ augmente.

Il existe au moins quatre façons de relaxer P: élargir l'ensemble des tuples autorisés par une contrainte, élargir le domaine d'une variable, retirer une contrainte et retirer une variable. On remarque que la deuxième et la troisième relaxation peuvent être faites en élargissant l'ensemble des tuples autorisés par une contrainte de $\mathcal{C}$. On peut en effet considérer le domaine d'une variable $x \in \mathcal{X}$ comme une contrainte unaire qui impose à x de prendre une valeur dans l'ensemble D_x. De plus, retirer une contrainte c peut être fait en élargissant l'ensemble des tuples autorisés par la contrainte c pour inclure le produit cartésien des domaines de $\mathcal{X}(c)$. Finalement, retirer une variable x correspond à autoriser pour chaque contrainte $c \in \mathcal{C}(x)$ les tuples qui satisfont c sans tenir compte de x. Par exemple, soit

une contrainte "toutes différentes" c agissant sur les variables x_1, x_2 et x_3 dont le domaine est $\{1, 2, 3\}$. Les tuples autorisés par c sont $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ et $(3, 2, 1)$, soit les permutations des valeurs du domaine des variables. Cependant, si on retire x_1 de c, l'ensemble des tuples autorisés devient $\{(*, 1, 2), (*, 1, 3), (*, 2, 1), (*, 2, 3), (*, 3, 1), (*, 3, 2)\}$, où "*" peut être n'importe quelle valeur de $\{1, 2, 3\}$. Il est à noter que retirer une variable d'un CSP correspond à retirer les contraintes impliquant cette variable uniquement si ces contraintes sont unaires ou binaires. Finalement, on constate que le MCSP et le MWCSP sont des cas spéciaux du MPCSP où le CSP sur-contraint original est rendu réalisable en retirant un ensemble de contraintes dont la somme des coûts est minimum.

1.1.2 Méthodes de résolution

La plupart des problèmes de satisfaction de contraintes sont très difficiles à résoudre. Ainsi, le CSP est reconnu comme un problème NP-complet, alors que le MCSP, le MWCSP et le PCSP sont des problèmes NP-difficiles. Une grande variété de méthodes ont été proposées pour résoudre ces problèmes (Kondrak, 1994; Tsang, 1993), formant deux classes principales: les méthodes complètes ou exactes, et les méthodes incomplètes, inexactes ou approchées. Dans le cas du CSP, les méthodes complètes explorent généralement l'espace de recherche en entier afin de trouver toutes les solutions au CSP ou détecter que le problème est irréalisable. Par ailleurs, dans le cas du MCSP, du MWCSP, du PCSP, et des autres problèmes de satisfaction NP-difficiles, le but des méthodes exactes est de trouver une solution optimale selon la fonction objectif. Ces méthodes de résolution, principalement basées sur des techniques de retour-arrière, de branch-and-bound et de filtrage (Freuder et Wallace, 1992; Larossa et Meseguer, 1995; Wallace, 1996), offrent une garantie de complétude, dans le cas des méthodes complètes, et d'optimalité, dans le cas des méthodes exactes. Par contre, comme la complexité de ces problèmes croît ex-

ponentiellement suivant leur taille, il est souvent impossible d'utiliser ce type de méthodes sur de plus grosses instances.

La seconde catégorie de méthodes, appelées méta-heuristiques, utilise principalement des heuristiques pour explorer certaines parties intéressantes de l'espace de recherche afin d'y trouver des solutions. Contrairement aux méthodes complètes ou exactes, ces méthodes ne permettent pas d'obtenir toutes les solutions, de détecter l'incohérence d'un CSP, et ne garantissent pas l'optimalité d'une solution. Généralement, les méta-heuristiques se basent sur des techniques de réparation, où une affectation initialement incohérente est itérativement modifiée afin de minimiser le nombre de contraintes violées ou la somme du coût de ces contraintes. Parmi ces méthodes, on trouve l'algorithme du moindre-conflit (MCH) (Wallace, 1996) pour résoudre le MCSP. Il existe également plusieurs raffinements du MCH, tel que le MCRW qui ajoute un déplacement aléatoire dans l'espace de recherche, et le MCTS (Galinier et Hao, 1997) qui ajoute une mémoire interdisant d'appliquer à nouveau une modification faite récemment à l'affectation. Ces raffinements servent à contourner le principal défaut du MCH, soit de se bloquer dans les minima locaux.

1.2 Sous-systèmes incohérents irréductibles

Une certaine forme d'incohérence survient fréquemment dans les problèmes réels. Cette incohérence peut provenir du modèle qui est physiquement irréalisable ou bien de l'ajout de contraintes supplémentaires qui ont rendu le problème surcontraint. Lorsque cela se produit, il est souvent difficile d'expliquer les causes de cette incohérence ou même de la régler. D'ailleurs, simplement détecter l'incohérence d'un CSP est un problème NP-complet que les méta-heuristiques ne permettent pas de résoudre. Cependant, dans la majorité des cas, l'incohérence est causée par des

contradictions locales du problème. Ainsi, un problème irréalisable contient souvent une sous-partie elle-même incohérente permettant de démontrer et d'expliquer l'incohérence du problème en entier. Suivant la terminologie dans (Carver, 1921; Chinneck, 1997 (1); Van Loon, 1981), un sous-système incohérent irréductible (IIS) d'un problème irréalisable est une sous-partie minimale (au sens de l'inclusion) de ce problème, elle-même irréalisable. Dans le cadre des problèmes de satisfaction de contraintes, un IIS peut être lié à un sous-ensemble de contraintes ou de variables:

Définition Étant donné un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ incohérent, un sous-ensemble de contraintes $K \subseteq \mathcal{C}$ est incohérent s'il n'existe aucune affectation de $\mathcal{X}$ satisfaisant toutes les contraintes de K. Un sous-ensemble de variables $W \subseteq \mathcal{X}$ est incohérent s'il n'existe aucune affectation de W satisfaisant toutes les contraintes de $\mathcal{C}$.

Définition Étant donné un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ incohérent, un sous-ensemble de contraintes $K \subseteq \mathcal{C}$ est un sous-ensemble incohérent irréductible (IIS) de contraintes s'il est incohérent, et si chaque sous-ensemble $K' \subset K$ est cohérent. De même, un sous-ensemble de variables $W \subseteq \mathcal{X}$ est un IIS de variables s'il est incohérent, mais chaque sous-ensemble $W' \subset W$ est cohérent.

Propriété 1.2.1 *Tout CN incohérent $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ contient au moins un IIS de contraintes $K \subseteq \mathcal{C}$, ainsi qu'au moins un IIS de variables $W \subseteq \mathcal{X}$.*

Propriété 1.2.2 *Étant donné un CN incohérent $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ et un IIS de contraintes $K \subseteq \mathcal{C}$, le sous-ensemble des variables impliquées dans les contraintes de K forment un IIS de variables. Par ailleurs, étant donné un IIS de variables $W \subseteq \mathcal{X}$, le sous-ensemble de contraintes agissant sur les variables de W ne forme pas nécessairement un IIS de contraintes (i.e est irréalisable, mais peut être réductible).*

1.3 Le coloriage de graphe

Le coloriage de graphe est un problème ayant un vaste ensemble d'applications, et pour lequel une grande variété de techniques de résolution a été proposée. Cette section présente le coloriage de graphe et le k-coloriage de graphe, ainsi que le lien de ces problèmes avec la satisfaction de contraintes et les IIS. Ces problèmes seront ensuite utilisés, dans le reste du mémoire, pour illustrer les notions présentées et tester les algorithmes dans la partie expérimentale.

1.3.1 Quelques définitions

Définition Étant donné un graphe $G = (V, E)$, où V est un ensemble de sommets et E un ensemble d'arêtes (i, j) reliant les sommets v_i et v_j, et un entier $k > 0$, une k-coloration de G est une fonction $r : V \rightarrow \{1, \ldots, k\}$ qui associe à chaque sommet $v \in V$ une couleur $r(v)$. Une k-coloration est dite valide si $r(v_i) \neq r(v_j)$ pour chaque arête $(i, j) \in E$. Le problème de k-coloriage de graphe pour G consiste à trouver une k-coloration valide des sommets de G ou à démontrer qu'une telle coloration n'existe pas.

Ce problème est directement relié au problème de coloriage de graphe:

Définition Étant donné un graphe G, le problème de coloriage de graphe pour G consiste à trouver le plus petit entier $k > 0$ tel qu'une k-coloration de G existe. La valeur minimale de k pour G correspond à son nombre chromatique $\chi(G)$.

Les problèmes de k-coloriage et de coloriage de graphe sont des problèmes respectivement NP-complet et NP-difficile (Garey et Johnson, 1979). Il est donc difficile

d'obtenir le nombre chromatique d'un graphe ayant un nombre élevé de sommets, sauf si le graphe possède une structure particulière permettant de déduire $\chi(G)$. De manière générale, le nombre chromatique d'un graphe augmente selon le nombre de sommets et sa densité, définie comme suit:

Définition Soit un graphe $G = (V, E)$, la densité d de G vaut:

$$d = \frac{2\ |E|}{|V|\ (|V| - 1)}$$

Ainsi, plus un graphe est gros et dense, plus son nombre chromatique risque d'être élevé. Inversement, un graphe ayant peu de sommets ou ayant une faible densité a un petit nombre chromatique. Il est ainsi possible de diminuer le nombre chromatique d'un graphe en lui retirant des sommets ou des arêtes. Un graphe est dit critique s'il est impossible de le réduire sans diminuer son nombre chromatique:

Définition Étant donné un graphe $G = (V, E)$, G est dit arête-critique si n'importe quel sous-graphe, produit en retirant une arête, a un nombre chromatique strictement inférieur à $\chi(G)$. De même, G est dit sommet-critique si n'importe quel sous-graphe, produit en retirant un sommet, a un nombre chromatique strictement inférieur à $\chi(G)$.

Propriété 1.3.1 *Tout graphe arête-critique est également sommet-critique. Par contre, un graphe sommet-critique n'est pas nécessairement arête-critique.*

Propriété 1.3.2 *Soit un graphe $G = (V, E)$ et un entier $k \leq \chi(G)$. G contient au moins un sous-graphe arête-critique et un sous-graphe sommet-critique de nombre chromatique k.*

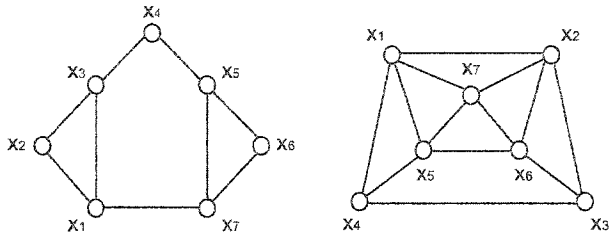

Figure 1.1 Exemples d'instance du k-coloriage de graphe

1.3.2 Modélisation du k-coloriage en CSP

Le problème de k-coloriage d'un graphe $G = (V, E)$ se modélise facilement en CSP en associant chaque sommet $v \in V$ à une variable $x \in \mathcal{X}$ de domaine $D_x = \{1, \ldots, k\}$, et chaque arête $(i, j) \in E$ à une contrainte binaire $c \in \mathcal{C}$ satisfaite uniquement lorsque $a(x_i) \neq a(x_j)$. Une k-coloration de V correspond alors à une affectation complète de $\mathcal{X}$. Par ailleurs, si G est arête-critique, le CSP correspondant au problème de k-coloriage de G forme un IIS de contraintes, si $k = \chi(G) - 1$. De la même manière, le CSP correspondant au problème de k-coloriage d'un graphe sommet-critique forme un IIS de variables, également si $k = \chi(G) - 1$.

1.3.3 Quelques exemples

Le figure 1.1 montre deux graphes de 7 sommets. Le graphe de gauche a 9 arêtes et un nombre chromatique de 3, car au moins trois couleurs sont nécessaires pour colorier ses sommets sans conflits. Le problème de 2-coloriage de ce graphe est donc irréalisable. On remarque que le CSP équivalent à ce problème contient quatre IIS de contraintes: $\{(1,2), (1,3), (2,3)\}$, $\{(5,6), (5,7), (6,7)\}$, $\{(1,3), (1,7), (3,4), (4,5), (5,7)\}$, et $\{(1,2), (1,7), (2,3), (3,4), (4,5), (5,6), (6,7)\}$. Ces IIS de contraintes sont composés des arêtes de quatre sous-graphes arête-critiques de nombre chromatique 3, ayant la forme de cycles impairs. De plus, le CSP ne

contient que trois IIS de variables: $\{x_1, x_2, x_3\}$, $\{x_5, x_6, x_7\}$ et $\{x_1, x_3, x_4, x_5, x_7\}$, formés des sommets de trois sous-graphes sommet-critiques pour $\chi(G) = 3$. Le nombre d'IIS de contraintes est nécessairement supérieur ou égal à celui de variables puisque chaque IIS de variables provient d'un IIS de contraintes. Dans cet exemple-ci, il y a un IIS de contraintes de plus, celui formé de 7 contraintes. Les variables impliquées dans ces contraintes ne peuvent pas être un IIS de variables puisque cet ensemble correspond à $\mathcal{X}$ qui est réductible. Par ailleurs, le graphe de droite est légèrement plus dense que celui de gauche (11 arêtes au lieu de 9), et son nombre chromatique est 4. On remarque, de plus, que ce graphe est sommet-critique, car si on lui retire n'importe quel sommet, il devient coloriable en 3 couleurs. Tel que mentionné dans la propriété 1.3.1, un graphe sommet-critique n'est pas nécessairement arête-critique. On peut ainsi retirer l'arête $(1, 2)$ de ce graphe sans diminuer $\chi(G)$.

1.4 Objectifs

Les sections précédentes ont permis d'introduire le cadre de la satisfaction de contraintes et la notion de sous-système irréalisable irréductible (IIS) d'un problème insoluble. Cependant, on ne possède, à ce jour, qu'un cadre théorique vague sur les IIS et leur relation avec la complexité de problèmes comme la satisfaction de contraintes. Par contre, des travaux portant sur le problème de satisfaction booléenne (SAT) (Mazure et al., 1998) et sur le coloriage de graphe (Herrmann et Hertz, 2002) ont montré que la recherche d'IIS dans un problème insoluble permettait d'aider la détection de son incohérence. De plus, des recherches dans les domaines de la programmation linéaire (Amaldi et al., 2003; Chinneck, 1997 (1); Parker et Ryan, 1996) et de la configuration de produits (Amilhastre et al., 2002) ont également montré l'utilité des IIS pour trouver les causes de l'incohérence d'un problème et

faciliter son diagnostic.

Bien que plusieurs méthodes permettant de trouver des IIS dans un problème insoluble aient été proposées, ces méthodes ne sont valides que pour un problème particulier. Le but de ce mémoire est de présenter des algorithmes efficaces pour trouver des IIS dans un cadre plus général, celui de la satisfaction de contraintes. Ces algorithmes peuvent alors être utilisés sur n'importe quel problème spécifique pouvant être modélisé en problème de satisfaction de contraintes.

1.5 Plan du mémoire

Ce mémoire renferme six chapitres. Suivant cette introduction, le chapitre 2 fait un survol des travaux précédents portant sur la détection d'IIS et leurs applications. Le chapitre 3 présente les algorithmes exacts de détection d'IIS. Le chapitre 4 offre, ensuite, une solution heuristique à la détection d'IIS. Le chapitre 5 contient l'expérimentation faite pour évaluer les algorithmes de détection et vérifier l'utilité de détecter des IIS. Finalement, le chapitre 6 fait une synthèse des travaux réalisés et propose de nouvelles voies de recherche.

CHAPITRE 2

HISTORIQUE DES IIS

2.1 IIS et la programmation linéaire

La notion de sous-système irréductible irréalisable (IIS) est d'abord apparue au début du vingtième siècle, dans le contexte de la programmation linéaire. La programmation linéaire permet de résoudre efficacement une catégorie de problèmes pouvant être traduits sous la forme d'un programme linéaire:

Définition Soit m et n deux entiers positifs, $A \in Z^{m \times n}$ une matrice de coefficients, $b \in Z^m$ et $c \in Z^n$ deux vecteurs de coefficients, et x un vecteur de m variables. Un programme linéaire (LP) est défini comme:

$$\begin{aligned} &\text{minimiser} \quad cx \\ &\text{sujet à} \quad\ Ax \leq b \end{aligned}$$

où cx est la fonction objectif et $Ax \leq b$ un système de contraintes d'inégalités linéaires. Un LP est réalisable s'il existe un vecteur x satisfaisant toutes les contraintes d'inégalités, sinon il est irréalisable.

Dans (Carver, 1921), Carver définit un IIS comme un ensemble incohérent de contraintes d'inégalités linéaires qui devient réalisable lorsque n'importe quelle contrainte est retirée. Une difficulté souvent rencontrée lors de la modélisation de gros LPs est l'identification d'erreurs et d'incohérence dans le modèle. Des travaux subséquents (Van Loon, 1981; Greenberg, 1992; Chinneck, 1997 (1)) ont cependant

montré qu'il était possible d'utiliser les IIS pour faciliter le diagnostic d'un LP irréalisable en isolant son incohérence. Dans (Van Loon, 1981). Van Loon propose une approche basée sur la géométrie du problème pour identifier des IIS. Cette technique a fourni une base pour le développement d'autres approches géométriques, comme celle proposée par Gleeson et Ryan (Gleeson et Ryan, 1990). Cette dernière méthode, sous certaines conditions (i.e. absence de dégénérescences dans le système d'équations), permet d'énumérer tous les IIS d'un LP irréalisable. Plus récemment, Chinneck a développé un ensemble d'algorithmes dont le but est d'identifier des IIS ayant le moins de contraintes possible (Chinneck, 1997 (1); Chinneck, 1997 (2)). L'idée de Chinneck est que moins un IIS contient de contraintes, plus son analyse est facile. Quelques uns de ces algorithmes sont décrits dans les trois prochaines sous-sections. Pour une description complète des algorithmes proposés par Chinneck, ainsi que pour obtenir les preuves aux propriétés énoncées, le lecteur est invité à consulter les références mentionnées ci-haut.

2.1.1 Filtre d'effacement

Le filtre d'effacement est un algorithme qui reçoit en paramètre un ensemble incohérent de contraintes linéaires C et qui retourne un IIS de contraintes K. Cet algorithme procède en retirant temporairement une contrainte choisie selon un ordre donné. Si l'ensemble réduit devient réalisable, cette contrainte est ré-insérée dans C, sinon elle est retirée de façon permanente.

Propriété 2.1.1 *Le filtre d'effacement retourne un ensemble de contraintes linéaires K formant un IIS.*

On remarque que l'ordre dans lequel les contraintes sont retirées détermine lequel IIS est détecté par le filtre d'effacement. Ainsi, tant qu'il reste un IIS dans C, cet

Algorithme 1 Filtre d'effacement

Entrée: Un ensemble incohérent de contraintes linéaires C;
Sortie : Un IIS de contraintes linéaires K.

```
Choisir un ordonnancement c_1, c_2, ..., c_|C| des contraintes de C;
pour i ← 1 à |C| faire
  K ← K \ {c_i};
  si K est réalisable alors
    K ← K ∪ {c_i};
  fin si
fin pour
```

ensemble est incohérent, et les contraintes retirées ne sont pas ré-insérées. L'IIS obtenu par le filtre d'effacement correspond donc au dernier IIS a avoir une contrainte retirée.

2.1.2 Algorithme additif

Alors que le filtre d'effacement procède en retirant des contraintes, l'algorithme additif proposé par Tamiz et al. (Tamiz et al., 1996) fait l'inverse. En commençant avec un ensemble de contraintes K vide, cet algorithme insère, selon un ordre donné, des contraintes dans un ensemble T initialement égal à K, jusqu'à ce que T devienne incohérent. L'ensemble T contient alors au moins un IIS dont fait partie la dernière contrainte insérée. Cette contrainte est alors rajoutée à l'IIS en construction K. Ce processus est répété jusqu'à ce que K devienne incohérent.

Propriété 2.1.2 *L'algorithme additif retourne un ensemble de contraintes linéaires K formant un IIS.*

Comme pour le filtre d'effacement, l'IIS obtenu par l'algorithme additif dépend de l'ordre initial des contraintes de C. Ainsi, l'algorithme additif se termine lorsque

Algorithme 2 Algorithme additif

Entrée: Un ensemble incohérent de contraintes linéaires C;
Sortie : Un IIS de contraintes linéaires K.

$K \leftarrow \emptyset$;
Choisir un ordonnancement $c_1, c_2, \ldots, c_{|C|}$ des contraintes de C;
répéter
  $T \leftarrow K$;
  $i \leftarrow 0$;
  tant que T est cohérent **faire**
    $T \leftarrow T \cup \{c_i\}$;
    $i \leftarrow i + 1$;
  fin tant que
  $K \leftarrow K \cup \{c_i\}$;
juqu'à ce que K soit incohérent

K forme un IIS, et la dernière contrainte ajoutée à K détermine quel IIS est détecté. L'IIS trouvé par l'algorithme additif est donc le premier à avoir sa dernière contrainte insérée dans K.

2.1.3 Filtre de sensibilité

Contrairement au filtre d'effacement et à l'algorithme additif, le filtre de sensibilité ne permet pas d'obtenir directement un IIS. Cet algorithme sert plutôt à accélérer l'isolation d'un IIS en retirant les contraintes ne contribuant pas à la fonction objectif, et, conséquemment, à l'incohérence du système. Ces contraintes peuvent ainsi être retirées car elles ne font pas partie de l'IIS étant isolé. Cette technique permet d'éliminer rapidement un grand nombre de contraintes du système original. Cependant, afin d'obtenir un IIS, il faut appliquer le filtre d'effacement ou l'algorithme additif au résultat du filtre de sensibilité.

2.1.4 Aspect qualitatif des IIS

Les algorithmes qui viennent d'être présentés permettent d'identifier tous les IIS de contraintes d'un LP irréalisable. Cependant, lequel de ces IIS est obtenu par ces algorithmes a un impact important sur la rapidité du diagnostic fait sur le système incohérent. Un LP possède deux types de contraintes: les contraintes de rangée, aussi appelées contraintes fonctionnelles, et les contraintes de colonne. Les contraintes de rangée sont des inéquations portant sur un ensemble de variables alors que les contraintes de colonne portent sur une seule variable. Des travaux réalisés par Greenberg (Greenberg, 1992) auprès d'usagers ont montré que les IIS contenant le moins de contraintes de rangée sont les plus utiles au diagnostic. Par exemple, soit un LP ayant deux IIS: le premier contenant 12 contraintes de rangée et 68 contraintes de colonne, et le second contenant une seule contrainte de rangée et 93 contraintes de colonnes. Bien que le premier IIS contienne, au total, moins de contraintes, le second IIS est, selon Greenberg, plus facile à interpréter et diagnostiquer. L'avantage des IIS ayant peu de contraintes de rangée provient du fait que les variables d'une contrainte de rangée sont presque toujours liées de manière complexe avec d'autres contraintes du système, ce qui rend leur interprétation difficile. De plus, minimiser le nombre de contraintes de rangée tend généralement à réduire le nombre de variables impliquées dans l'IIS obtenu.

Une approche triviale pour trouver l'IIS contenant le moins de contraintes de rangée est d'utiliser une méthode comme le filtre d'effacement ou l'algorithme additif pour énumérer tous les IIS du modèle, et d'ensuite choisir celui possédant le moins de ces contraintes. Cependant, il a été démontré par Chakravarti (Chakravarti, 1994) que le nombre d'IIS dans un LP irréalisable est, dans le pire cas, exponentiel. Cela signifie qu'en général un IIS ayant le moins de contraintes de rangée ne peut être obtenu en temps polynomial par une simple méthode d'énumération. Chinneck propose

donc d'utiliser des méthodes heuristiques pour trouver des IIS contenant le moins possible de ces contraintes, sans toutefois garantir l'optimalité. L'idée proposée par Chinneck est simple: comme l'ordre initial des contraintes de C détermine quel IIS est détecté dans le filtre d'effacement ou l'algorithme additif, il suffit d'utiliser une heuristique favorisant le retrait ou l'ajout de certaines contraintes du système. Par exemple, dans le cas du filtre d'effacement, il est préférable de retirer d'abord les contraintes de rangée de C. Par contre, dans l'algorithme additif, on ajoute à K les contraintes de colonne en premier. Des expériences faites sur des LPs de référence ont montré que les méthodes utilisant une telle heuristique permettent de trouver, dans la plupart des cas, des IIS ayant bien moins de contraintes de rangée que ceux obtenus par les méthodes sans heuristique.

2.1.5 Réparation de modèles irréalisables

Alors que détecter des IIS permet de faciliter le diagnostic d'un modèle de programmation linéaire irréalisable, rendre ce modèle à nouveau cohérent reste une tâche complexe. Il est cependant possible d'utiliser l'information provenant de plusieurs IIS pour automatiser cette tâche. Dans (Parker et Ryan, 1996), Parker et Ryan proposent d'utiliser les IIS d'un LP incohérent pour identifier le sous-système réalisable de poids maximum, dont suit la définition:

Définition Étant donné un système irréalisable d'inégalités linéaires C, et $w : C \rightarrow \mathbb{R}^+$ une fonction qui associe un poids positif $w(c)$ à chaque contrainte $c \in C$, le problème du sous-système réalisable de poids maximum (MWFS) correspond à trouver un sous-système $F \subset C$ réalisable qui maximise la fonction

$$h(F) = \sum_{c \in F} w(c)$$

correspondant à la somme des poids des contraintes de F.

Le MWFS est un problème d'optimisation NP-difficile qui généralise le problème du sous-système réalisable maximum (MFS), où les contraintes ont toutes le même poids. En permettant à l'usager d'affecter des poids différents aux contraintes selon leur importance dans le système, le sous-système réalisable de poids maximum correspond à une version réalisable du modèle la plus fidèle aux conditions originales posées par l'usager. Par ailleurs, le MWFS est équivalent au problème de recouvrement d'IIS de poids minimum:

Définition Étant donné un système irréalisable C d'inégalités linéaires, une fonction $w : C \rightarrow \mathbb{R}^+$ qui associe un poids positif $w(c)$ à chaque contrainte $c \in C$, et $\mathcal{I}$ l'ensemble des IIS de C, le problème de hitting set d'IIS de poids minimum (MWIC) consiste à trouver un sous-ensemble d'inégalités $H \subseteq C$ qui interesecte chaque IIS de $\mathcal{I}$ et minimise la fonction

$$g(H) = \sum_{c \in H} w(c)$$

correspondant à la somme des poids de H.

Le MWIC est également un problème d'optimisation NP-difficile qui généralise le problème NP-difficile du *hitting set* de cardinalité minimum (MHS) (Garey et Johnson, 1979) où tous les éléments ont le même poids:

Définition Soit $\mathcal{E} = \{E_1, \ldots, E_{|\mathcal{E}|}\}$ une collection d'ensembles $E_i \subseteq S, 1 \leq i \leq |\mathcal{E}|$. Un ensemble $H \subseteq S$ est un *hitting set* (HS) de $\mathcal{E}$ s'il intersecte chaque ensemble $E_1, \ldots, E_{|\mathcal{E}|}$. Le problème de *hitting set* minimum (MHS) pour une collection $\mathcal{E}$ consiste à déterminer un *hitting set* de cardinalité minimum de $\mathcal{E}$.

Une représentation équivalente du MWIC est de trouver un ensemble transversal de cardinalité minimum dans un hyper-graphe où chaque hyper-arête correspond à un IIS de $\mathcal{I}$ et les sommets appartenant à cette hyper-arête correspondent aux contraintes de l'IIS:

Définition Soit un hyper-graphe $G = (V, E)$. Un ensemble de sommets $T \subseteq V$ est transversal si chaque hyper-arête $e \in E$ contient un sommet de T.

Le MWIC est un problème équivalent au MWFS puisque, pour rendre réalisable un problème sur-contraint, il suffit de relaxer au moins une contrainte de chaque IIS de $\mathcal{I}$. Soit un *hitting set* H des IIS de $\mathcal{I}$, l'ensemble complémentaire de contraintes (i.e. $C \setminus H$) est donc nécessairement cohérent. Par ailleurs, puisque

$$\sum_{c \in C} w(c) = \sum_{c \in H} w(c) + \sum_{c \in C \setminus H} w(c)$$

minimiser la somme des poids des contraintes de H correspond à maximiser la somme des poids des contraintes de l'ensemble complémentaire. Ainsi, étant donné un ensemble de contraintes optimal à une instance de MWIC, l'ensemble complémentaire est optimal au MWFS correspondant.

Résoudre une instance du MWIC suppose que l'on connaisse l'ensemble $\mathcal{I}$ des IIS du problème. Cependant, comme démontré dans (Chakravarti, 1994) un système irréalisable de contraintes peut contenir un nombre exponentiel d'IIS. Il est donc impossible, en pratique, de résoudre le MWFS de manière optimale en utilisant le MWIC. Dans (Parker, 1995), Parker propose donc d'utiliser une approche itérative pour résoudre le MWIC.

Cet algorithme débute avec un ensemble d'IIS $\mathcal{I}$ vide et un *hitting set* H vide. Par la suite, tant qu'il existe un IIS K non-couvert par H, cet IIS est ajouté à $\mathcal{I}$ et

Algorithme 3 Algorithme itératif pour le MWIC

Entrée: Un ensemble incohérent de contraintes linéaires C;
Sortie : Un recouvrement optimal H des IIS de C.

$\mathcal{I} \leftarrow \emptyset$;
$H \leftarrow \emptyset$;
Continuer $\leftarrow$ VRAI;
tant que Continuer = VRAI **faire**
  si $\exists$ un IIS $K \subseteq C$ non couvert par H **alors**
    $I \leftarrow I \cup \{K\}$;
    $H \leftarrow$ procMWHS($\mathcal{I}$);
  sinon
    Continuer $\leftarrow$ FAUX
  fin si
fin tant que

un nouveau *hitting set* de poids minimum (MWHS) est obtenu par la procédure *procMWHS*. Ce processus est répété jusqu'à ce que tous les IIS de C se trouvent dans $\mathcal{I}$. Il est également possible d'arrêter l'algorithme à n'importe quelle étape, auquel cas H est une solution approchée au MWIC. Par ailleurs, comme le MWHS est un problème NP-difficile, *procMWHS* peut être remplacée par une méthode heuristique, au risque d'avoir un recouvrement H sous-optimal. Finalement, l'IIS non-couvert par H peut être obtenu à l'aide d'une approche géométrique, comme celles proposées par Van Loon(Van Loon, 1981), par Gleeson et Ryan(Gleeson et Ryan, 1990), ou grâce à un algorithme de détection proposé par Chinneck.

2.2 IIS et le support à la décision

Dans (Amilhastre et al., 2002), on propose d'utiliser les IIS pour résoudre des problèmes de support à la décision interactive, où la tâche est d'assister l'usager dans le choix de valeurs pour les variables du système qui satisfont les contraintes de ce système. Un exemple de ce type de problème, la configuration de produits

(Stumptner, 1997), peut être représenté par un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ où les variables $\mathcal{X}$ sont les composantes ou les options d'un certain produit, et l'ensemble des contraintes $\mathcal{C}$ représente la compatibilité entre les composantes, leur disponibilité, etc. L'ensemble des solutions du CSP correspondant représente le catalogue, c'est-à-dire toutes les variantes offertes de ce produit. Durant la configuration du produit, l'usager spécifie de façon interactive un ensemble contraintes additionnelles $\{c_1, c_2, \ldots\}$ représentant ses préférences. À une certaine itération p, l'ajout d'une contrainte c_p peut soudainement rendre incohérent l'ensemble des contraintes de configuration (i.e. $C \cup \{c_1, \ldots, c_p\}$ est incohérent alors que $C \cup \{c_1, \ldots, c_{p-1}\}$ ne l'est pas). Dans ce cas, le système de support à la décision doit guider l'usager dans le choix des contraintes à relaxer afin de rendre le modèle à nouveau réalisable, et fournir à l'usager une explication sur l'incohérence. Suivant la terminologie utilisée dans (Amilhastre et al., 2002), voici quelques définitions importantes:

Définition Soit $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, H)$ un 4-tuple où $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est un CN représentant un produit configurable, et H un ensemble fini de contraintes additionnelles sur $\mathcal{X}$, correspondant aux restrictions de l'usager. Un *nogood* de P est un ensemble $E \subseteq H$ de cardinalité minimale, tel que $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup E)$ est incohérent. De la même manière, une *interprétation* de $\mathcal{P}$ est un ensemble $E \subseteq H$ de cardinalité maximale, tel que $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup E)$ est cohérent.

Un nogood est relié à une interprétation par la propriété suivante:

Propriété 2.2.1 *Soit N l'ensemble des nogoods de $\mathcal{P}$, E est une interprétation de $\mathcal{P}$ si et seulement si $H \setminus E$ est un* hitting set *de N de cardinalité minimale.*

Les notions de nogood et d'interprétation sont directement reliées à celles de MWIC et MWFS puisque qu'un nogood est en fait un IIS de contraintes et une interprétation un sous-système réalisable de cardinalité maximale, dans le cas où

les contraintes ont toutes le même poids. De plus, les problèmes de recouvrement d'ensemble et celui d'*hitting sets* sont en fait des problèmes équivalents. Afin d'illustrer ces notions, prenons le problème de configuration d'une automobile où les variables sont la couleur de certaines pièces de l'automobile: $\mathcal{X} = \{$*pare-chocs, toit, roues, carrosserie, capot, portes*$\}$. Chacune de ces pièces doivent avoir une des couleurs suivantes: $\{$*blanc, rose, rouge, noir*$\}$. Par ailleurs, ces pièces sont soumises aux contraintes de configurations $\mathcal{C} = \{c_1, \ldots, c_6\}$, chacune portant sur les variables suivantes: $\mathcal{X}(c_1) = \{$*carrosserie, portes*$\}$, $\mathcal{X}(c_2) = \{$*capot, portes*$\}$, $\mathcal{X}(c_3) = \{$*carrosserie, capot*$\}$, $\mathcal{X}(c_4) = \{$*pare-chocs, carrosserie*$\}$, $\mathcal{X}(c_5) = \{$*toit, carrosserie*$\}$, $\mathcal{X}(c_6) = \{$*roues, carrosserie*$\}$. De plus, l'ensemble des affectations autorisées des variables de ces contraintes sont: $\mathcal{R}(c_1) = \mathcal{R}(c_2) = \mathcal{R}(c_3) = \{$(*blanc, blanc*), (*rose, rose*), (*rouge, rouge*), (*noir, noir*)$\}$, $\mathcal{R}(c_4) = \mathcal{R}(c_5) = \mathcal{R}(c_6) = \{$(*blanc, rose*), (*blanc, rouge*), (*blanc, noir*), (*rose, rouge*), (*rose, noir*), (*rouge, noir*)$\}$. L'ensemble des restrictions de l'usager est: $H = \{h_{\text{pare-chocs}}, h_{\text{toit}}, h_{\text{roues}}, h_{\text{carrosserie}}, h_{\text{capot}}, h_{\text{portes}}\}$. Ces contraintes portent sur une seule pièce de l'automobile (*dont le nom est en indice*), et imposent à ces pièces d'avoir les couleurs suivantes: $\mathcal{R}(h_{\text{pare-chocs}}) = \mathcal{R}(h_{\text{toit}}) = \{$*blanc, rose*$\}$, $\mathcal{R}(h_{\text{roues}}) = \{$*rouge*$\}$, $\mathcal{R}(h_{\text{carrosserie}}) = \{$*rose, rouge*$\}$, $\mathcal{R}(h_{\text{portes}}) = \{$*rouge, noir*$\}$, $\mathcal{R}(h_{\text{capot}}) = \{$*rose, noir*$\}$. Cet exemple comporte deux nogoods: $\{h_{\text{carrosserie}}, h_{\text{capot}}, h_{\text{portes}}\}$ et $\{h_{\text{roues}}, h_{\text{carrosserie}}\}$. Ainsi, le dernier ensemble est un nogood parce que h_{roues} impose d'avoir des roues rouges et $h_{\text{carrosserie}}$ d'avoir une carrosserie rouge ou rose, alors que la contrainte c_6 n'autorise aucune de ces deux combinaisons. Par ailleurs, l'exemple comporte trois interprétations: $\{h_{\text{pare-chocs}}, h_{\text{toit}}, h_{\text{roues}}, h_{\text{capot}}, h_{\text{portes}}\}$, $\{h_{\text{pare-chocs}}, h_{\text{toit}}, h_{\text{carrosserie}}, h_{\text{portes}}\}$ et $\{h_{\text{pare-chocs}}, h_{\text{toit}}, h_{\text{carrosserie}}, h_{\text{capot}}\}$. Ces ensembles sont ainsi cohérents avec les contraintes de $\mathcal{C}$, mais l'ajout de n'importe quelle autre contrainte de H rendrait cet ensemble incohérent avec $\mathcal{C}$.

Lorsque l'ensemble H des restrictions de l'usager n'est pas cohérent avec l'ensemble $\mathcal{C}$ des contraintes de configuration du produit, le système doit pouvoir fournir à l'usager une *explication* sur l'incohérence:

Définition Soit $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, H)$ un 4-tuple, et c une contrainte de H, une explication de c sur P est un ensemble $E \subseteq H$ tel que $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup E)$ est cohérent tandis que $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup E \cup \{c\})$ est incohérent. Une explication est minimale s'il n'existe aucune autre explication $E' \subset E$. Par ailleurs, une restauration de c sur P est un ensemble $E \subseteq H$ tel que $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup E \cup \{c\})$ est cohérent. Une restauration E est maximale s'il n'existe aucune autre restauration $E' \supset E$.

Une explication permet donc d'expliquer pourquoi la contrainte rajoutée c rend le modèle irréalisable, alors qu'une restauration est un sous-ensemble de contraintes permettant de rendre le modèle à nouveau réalisable, tout en conservant c. En revenant à l'exemple précédent de configuration d'une automobile, supposons que l'ensemble des restrictions de l'usager soit:
$H = \{h_{\text{pare-chocs}}, h_{\text{toit}}, h_{\text{roues}}, h_{\text{capot}}, h_{\text{portes}}\}$, avec $\mathcal{R}(h_{\text{pare-chocs}}) = \mathcal{R}(h_{\text{toit}}) = \{\textit{blanc, rose}\}$, $\mathcal{R}(h_{\text{roues}}) = \{\textit{rouge}\}$, $\mathcal{R}(h_{\text{portes}}) = \{\textit{blanc, rouge, noir}\}$, $\mathcal{R}(h_{\text{capot}}) = \{\textit{rose, noir}\}$. Soit la contrainte supplémentaire c, telle que $\mathcal{X}(c) = \{\textit{carrosserie}\}$ et $\mathcal{R}(c) = \{\textit{rose}\}$, imposant à la carrosserie d'être rose. Cette contrainte a deux explications minimales: $\{h_{\text{roues}}\}$ et $\{h_{\text{portes}}\}$. Ainsi, h_{roues} est une explication car cette contrainte est cohérente avec $\mathcal{C}$, mais devient incohérente lorsqu'on rajoute c (i.e. c_6 n'autorise pas d'avoir à la fois les roues rouges et la carrosserie rose). Par ailleurs, si on veut rendre ce problème réalisable tout en conservant c, il faut relaxer h_{roues} et h_{portes}. Ainsi, $\{h_{\text{pare-chocs}}, h_{\text{toit}}, h_{\text{capot}}\}$ est une restauration maximale de c pour le modèle.

2.3 IIS et SAT

Les IIS peuvent également servir à prouver qu'un problème est irréalisable. Ainsi, un problème localement incohérent contient une sous-partie de taille réduite qui est elle-même incohérente. Par ailleurs, pour démontrer de manière exacte que le problème est irréalisable, il suffit de prouver que la sous-partie est incohérente. Cette technique a été utilisée par Mazure et al. (Mazure et al., 1998) sur le problème de satisfaction booléenne (SAT):

Définition Une formule booléenne de la forme normale conjonctive (FNC) est une conjonction de clauses formées par une disjonction de littéraux. Un littéral est une variable propositionnelle pouvant avoir une valeur de vérité positive ou négative.

Définition Étant donné une formule booléenne sous la forme FNC, le problème de *satisfaction booléenne* (SAT) consiste à déterminer s'il existe une affectation de valeurs aux variables vérifiant la formule booléenne.

Une instance de SAT se traduit en CSP en associant chaque variable du SAT à une variable de $\mathcal{X}$, et chaque clause à une contrainte de $\mathcal{C}$, en respectant la valeur de vérité des variables dans les littéraux. Par ailleurs, SAT est reconnu pour être NP-complet (Cook, 1971). Il existe ainsi plusieurs techniques efficaces pour résoudre une instance de SAT de manière exacte, tel que l'algorithme *DP* proposé par Davis et Putnam (Davis et Putnam, 1960). Cependant, ces techniques ne permettent pas, en général, de résoudre des instances incohérentes possédant un grand nombre de variables et de clauses. Par contre, les méthodes heuristiques, principalement basées sur le principe de recherche locale, permettent généralement de trouver une solution aux grandes instances satisfaisables. L'algorithme GSAT, développé par Selman et al. (Selman et al., 1992), est une telle méthode heuristique.

Dans (Mazure et al., 1998), Mazure et al. utilisent une méthode heuristique pour aider une méthode exacte à démontrer qu'une instance de SAT est irréalisable en concentrant la méthode exacte sur une sous-partie incohérente de l'instance. À chaque fois que l'heuristique est utilisée, une trace de l'exécution est enregistrée de la façon suivante. Un compteur est associé à chaque clause et chaque littéral de l'instance. Ces compteurs sont d'abord initialisés à zéro. À chaque itération, la valeur d'une des variables est inversée dans l'affectation. Ensuite, les compteurs des clauses falsifiées par l'affectation, ainsi que ceux des littéraux de ces clauses, sont incrémentés de 1. Après un nombre donné d'itérations, la recherche se termine et la valeur de chaque compteur est examinée. Si le nombre d'itérations est suffisamment grand, les clauses de l'instance devraient pouvoir être séparées en deux ensembles: les clauses qui n'ont jamais ou rarement été falsifiées, et celles qui ont été souvent falsifiées. Intuitivement, le second ensemble renferme les clauses d'une ou plusieurs sous-parties incohérentes de l'instance. Considérant le CSP équivalent à l'instance SAT, les clauses du second ensemble sont un ensemble de contraintes approchant la réunion des *hitting sets* minimum des IIS de contraintes du CSP. Dans le cas où les IIS sont tous disjoints, cet ensemble estime l'union de tous les IIS du CSP. Par ailleurs, si l'intersection de tous les IIS n'est pas vide, le second ensemble estime cette intersection.

L'information des compteurs peut également être utilisée pour guider la stratégie de branchement de la méthode exacte, en sélectionnant d'abord les littéraux dont le compteur contient la plus grande valeur. Cette stratégie est basée sur le principe d'échec d'abord (*fail first principle*) qui permet d'éliminer rapidement de l'arbre de recherche les branches ne menant à aucune solution. Ainsi, les littéraux faisant partie des clauses les plus souvent falsifiées risquent de mener plus rapidement à un échec, éliminant le besoin d'explorer les sous-branches de l'affectation courante. Les expériences faites par Mazure et al. ont démontré qu'une telle stratégie permettait

à la méthode exacte de résoudre certaines instances réalisables et irréalisables de SAT plus rapidement (i.e. moins de branchements) qu'en utilisant une stratégie de branchement différente.

2.4 IIS et le coloriage de graphe

La section précédente présente une technique utilisant les IIS pour améliorer l'efficacité d'une méthode exacte pour le problème SAT. Cette section présente une utilisation différente des IIS pour résoudre de manière exacte le problème de coloriage de graphe, correspondant à trouver le nombre chromatique $\chi(G)$ d'un graphe G. Ce problème étant NP-difficile, les méthodes exactes ne parviennent qu'à trouver le nombre chromatique de graphes possédant un petit nombre de sommets. Ainsi, sur des graphes aléatoires dont la probabilité d'existence d'une arête entre chaque paire de sommets (i.e. la densité) est 0.5, les meilleures méthodes exactes, telles que (Kubale et Jackowski, 1985; Mehotra et Trick, 1996; Peemöller, 1983), sont généralement incapables de démontrer le nombre chromatique de graphes n'ayant que 100 sommets. En revanche, il est possible d'utiliser des méthodes heuristiques (Fleurent et Ferland, 1996; Galinier et Hao, 1999) sur de plus grosses instances, mais seulement pour obtenir une borne supérieure sur $\chi(G)$.

Dans (Herrmann et Hertz, 2002), Herrmann et Hertz proposent un nouvel algorithme qui utilise les graphes critiques pour résoudre de manière exacte le problème de coloriage de graphe. Étant donné un graphe G, cet algorithme calcule d'abord avec une méthode de coloration heuristique une borne $k \geq \chi(G)$. L'algorithme tente ensuite d'obtenir pour G un sous-graphe sommet-critique H de même nombre chromatique, et détermine $\chi(H)$ à l'aide d'une méthode exacte. Puisque H contient, en général, moins de sommets et d'arêtes que G, la méthode exacte, de complexité exponentielle, a plus de chances de pouvoir calculer, dans un temps

raisonnable, $\chi(H)$ que $\chi(G)$. Sachant que $\chi(H) \leq \chi(G)$, $\chi(G)$ est alors démontré si $k = \chi(H)$.

L'algorithme 4 donne les détails de cette stratégie. Cet algorithme prend en paramètre un graphe G et retourne $\chi(G)$. De plus, l'algorithme utilise deux procédures, *hCol* et *xCol*, qui sont, respectivement, des méthodes heuristiques et exactes pour obtenir le nombre chromatique d'un graphe. Le calcul de $\chi(G)$ se fait en deux phases. La phase descendante obtient tout d'abord une borne $k \geq \chi(G)$ à l'aide de *hCol*. Ensuite les sommets de H, initialement égal à G, sont retirés dans un certain ordre. Après chaque retrait, *hCol* calcule une borne supérieure sur $\chi(H)$. Si cette valeur est inférieure à k, le dernier sommet est remis dans H. Lorsque tous les sommets ont été ainsi testés, $k' = \chi(H)$ est obtenu à l'aide de *xCol*. Si $k' = k$, l'algorithme vient de trouver un sous-graphe sommet-critique H et de démontrer que $\chi(H) = \chi(G)$. Par contre, si $k' < k$, il s'est produit au moins une des erreurs suivantes: *hCol* a obtenu une borne $k > \chi(G)$, ou bien trop de sommets ont été retirés de H, tel que $\chi(H) < \chi(G)$. La phase augmentante de l'algorithme, qui permet de traiter ces cas, obtient l'ensemble L des sommets retirés qui augmenteraient $\chi(H)$ s'ils étaient remis dans H. Si L est vide, cela signifie que $k > \chi(G)$, et que $k' = \chi(G)$. L'algorithme se termine, et H n'est pas un sous-graphe sommet-critique. Par contre, si L n'est pas vide, un sommet est choisi de L et est remis dans H, tel que $\chi(H)$ augmente de 1. Ce processus est répété jusqu'à ce que $k' = k$, ou L soit vide. Dans le premier cas, l'algorithme vient de trouver un sous-graphe sommet-critique H et démontrer $\chi(H) = \chi(G)$.

Étant donné un graphe G, un sous-graphe sommet-critique H de G, est un IIS de variables dans le CSP correspondant au problème de coloriage de G avec $\chi(H) - 1$ couleurs (*voir section 1.3.2*). L'algorithme proposé par Herrmann et Hertz permet donc d'obtenir, si $k = \chi(G)$, des IIS de variables du CSP équivalent au k-coloriage de G. Par ailleurs, comme cet algorithme applique une méthode exacte pour

Algorithme 4 Algorithme de Herrmann et Hertz pour calculer $\chi(G)$

Entrée: Un graphe $G = (V, E)$;
Sortie : $\chi(G)$.

Phase descendante
Générer un graphe $H = (V', E')$ égal à G;
$k \leftarrow \text{hCol}(G)$;
Choisir un ordonnancement $v_1, v_2, \ldots, v_{|V|}$ des sommets de V;
pour $i \leftarrow 1$ à $|V|$ **faire**
  Retirer v_i de V';
  si $\text{hCol}(H) < k$ **alors**
    Remettre v_i dans V';
  fin si
fin pour
$k' \leftarrow \text{xCol}(H)$;

Phase augmentante
si $k' < k$ **alors**
  répéter
    $L \leftarrow \emptyset$;
    pour tout sommet $v \in V \setminus V'$ **faire**
      Remettre v dans V';
      si $\text{hCol}(H) > k'$ **alors**
        si $\text{xCol}(H) = k' + 1$ **alors**
          $L \leftarrow L \cup \{v\}$;
        fin si
      fin si
      Retirer v de V';
    fin pour
    si $L \neq \emptyset$ **alors**
      Choisir un sommet $v \in L$;
      Remettre v dans V';
      $k' \leftarrow k' + 1$;
    sinon
      STOP: $k > k' = \chi(G)$;
    fin si
  juqu'à ce que $k' = k$
fin si
$\chi(G) = k'$;

déterminer $\chi(H)$, il est nécessaire d'obtenir un sous-graphe H contenant le moins de sommets possible. Ainsi, les auteurs présentent une stratégie qui détermine l'ordre de retrait des sommets afin d'obtenir de plus petits sous-graphes H: le sommet ayant le plus faible degré (i.e. le moins de sommets adjacents) dans le graphe restant est d'abord retiré. Cette stratégie permet d'obtenir des sous-graphes H possédant moins de sommets. Finalement, des expériences, faites sur des graphes aléatoires et des instances provenant du second challenge DIMACS (Johnson et Trick, 1996), ont montré que, dans la plupart des cas, une méthode exacte (Peemöller, 1983) mettait moins de temps (i.e. moins de retour-arrières) à déterminer le nombre chromatique du sous-graphe critique que de l'instance originale.

2.5 IIS et le problème de recouvrement d'ensembles

Dans (Galinier et Hertz, 2003) Galinier et Hertz montrent que les problèmes de détection d'IIS de contraintes et de variables dans un CSP incohérent sont deux cas spéciaux du problème de recouvrement d'ensembles de grande taille ($LSCP$).

Définition Soit S un ensemble fini et $\mathcal{E} = \{E_1, \ldots, E_{|\mathcal{E}|}\}$, une collection d'ensembles $E_i \subseteq S$ dont l'union vaut S. Un sous-ensemble $I \subseteq \{1, \ldots, |\mathcal{E}|\}$ tel que $\bigcup_{i \in I} E_i = S$ est appelé recouvrement de S. Le problème de recouvrement d'ensembles à coût fixe ($USCP$) pour S correspond à trouver un recouvrement de cardinalité minimum de S.

Définition Soit S un ensemble fini et $\mathcal{E} = \{E_1, \ldots, E_{|\mathcal{E}|}\}$, une collection d'ensembles $E_i \subseteq S$ dont l'union vaut S. Soit également une fonction $\varphi : S \times \mathcal{E} \rightarrow \{0, 1\}$ telle que $\varphi(e, E)$ vaut 1 si $e \in E$, et 0 sinon. Finalement, soit une fonction $w : \mathcal{E} \rightarrow \mathbb{R}^+$ qui associe à chaque $E \in \mathcal{E}$ un poids positif $w(E)$, et une procédure $MinW(S, \mathcal{E}, w)$

qui retourne un élément $e \in E$ minimisant la fonction suivante:

$$f(e) = \sum_{E \in \mathcal{E}} \varphi(e, E)\ \omega(E)$$

Le LSCP consiste à déterminer, à l'aide de φ et $MinW$, un recouvrement minimal de S. Par ailleurs, le problème de recouvrement minimum d'ensembles de grande taille ($MLSCP$) consiste à déterminer, à l'aide de φ et $MinW$, un recouvrement minimum de S.

Le LSCP est une variante de l'USCP pour laquelle l'ensemble S et les sous-ensembles $E \in \mathcal{E}$ sont possiblement de très grande taille, et ne peuvent pas être donnés en extension. Ces ensembles sont plutôt définis implicitement à l'aide de la fonction φ qui permet de savoir si un élément e fait partie de E. De plus, le MLSCP est une extension du LSCP pour laquelle le but est de trouver un recouvrement de cardinalité minimum.

La détection d'IIS de contraintes et de variables dans un CSP incohérent peuvent être considérés comme des cas particuliers du LSCP. De même, la détection d'IIS minimum de contraintes et de variables sont des cas particuliers du MLSCP. Ainsi, dans le cas de la détection d'IIS de contraintes, on considère S comme l'ensemble de toutes les affectations complètes des variables de $\mathcal{X}$, et un sous-ensemble $E_i \in \mathcal{E}$ comme la contrainte $c_i \in \mathcal{C}$. Un élément e appartient alors à E_i si et seulement si l'affectation complète correspondant à e viole la contrainte c_i associée à E_i. Trouver un recouvrement minimal de S correspond ainsi à détecter un IIS de contraintes, c'est-à-dire un ensemble minimal de contraintes interdisant toute affectation des variables du CSP. De plus, dans le contexte des CSP $\varphi(e, E_i)$ vaut 1 si l'affectation correspondant à e viole la contrainte c_i, et vaut 0 sinon. Par ailleurs, w est une fonction qui pondère les contraintes du CSP, et $MinW$ est un algorithme qui trouve une affectation complète dont la somme des poids des contraintes violées est mini-

mum. Par contre, dans le contexte de la détection d'IIS de variables, S représente l'ensemble des affectations partielles légales des variables de $\mathcal{X}$, alors qu'un sous-ensemble $E_i \in \mathcal{E}$ est l'ensemble des affectations partielles légales pour lesquelles la variable x_i n'a pas de valeur (i.e. n'est pas instanciée). Ainsi, détecter un IIS de variables dans un CSP incohérent correspond à trouver un ensemble minimal de variables tel que chaque affectation partielle légale des variables du CSP interdise au moins à une de ces variables d'être instanciée. Dans cette optique, $\varphi(e, E_i)$ vaut 1 si et seulement si la variable x_i n'est pas instanciée dans l'affectation partielle correspond à e, et vaut 0 sinon. De plus, w est une fonction qui associe un poids à chaque variable du CSP, alors que $MinW$ est un algorithme qui trouve une affectation partielle légale telle que la somme des poids des variables n'étant pas instanciées est minimum.

Galinier et Hertz présentent également dans (Galinier et Hertz, 2003) plusieurs algorithmes pour résoudre le LSCP et le MLSCP, ainsi que pour calculer une borne inférieure sur la taille d'un recouvrement minimum. Ces algorithmes peuvent être utilisés pour faire la détection d'IIS de contraintes et de variables dans un CSP incohérent, ainsi que pour trouver des IIS minimum et une borne sur la taille de ces IIS. Le prochain chapitre présente une adaptation des algorithmes développés par Galinier et Hertz à la détection d'IIS dans un CSP incohérent. Ces algorithmes ont, par ailleurs, servi de base à l'élaboration d'autres techniques de détection qui seront également présentées dans le chapitre suivant.

CHAPITRE 3

ALGORITHMES EXACTS DE DÉTECTION D'IIS

Ce chapitre présente diverses techniques pour trouver des IIS de contraintes et de variables dans un CSP irréalisable, ainsi que pour obtenir des IIS de cardinalité minimum et des bornes sur la taille de ces IIS. Les algorithmes proposés par Galinier et Hertz pour résoudre le LSCP et le MLSCP (Galinier et Hertz, 2003), l'*algorithme de retrait*, l'*algorithme d'insertion* et l'*algorithme par hitting set*, sont d'abord décrits dans le contexte des CSP. Alors que ces algorithmes permettent tous d'obtenir des IIS de contraintes et de variables, seul l'algorithme de *hitting set* permet de trouver des IIS minimum. Cet algorithme sert également à obtenir une borne inférieure sur la taille d'un IIS minimum. Finalement, ce chapitre présente des variantes originales de ces algorithmes, ainsi que des heuristiques, également développées dans le cadre de ce mémoire, pour détecter des IIS possédant le moins de contraintes et de variables possible.

3.1 Notions préalables

La section 1.1 de ce mémoire donne une définition générale de satisfaction de contraintes pour les CN pondérés. Dans cette définition, une contrainte $c \in \mathcal{C}$ est une fonction qui associe, pour chaque affectation a des variables de $\mathcal{X}$, un coût $c(a)$. Bien que flexible, cette définition est trop complexe pour la description des algorithmes de détection d'IIS présentés dans ce mémoire. Cette section propose une définition simplifiée de la satisfaction de contraintes pour les CN pondérés, utilisant la définition classique d'une contrainte $c \in \mathcal{C}$ (i.e. fonction telle que $c(a)$ vaut 0

si c est satisfaite, et 1 sinon). Afin de pouvoir donner une importance différente à chaque contrainte et variable du CN, cette définition introduit également deux fonctions qui associent un poids à ces contraintes et variables.

Définition Soit un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ et $w : \mathcal{C} \rightarrow \mathbb{R}^+$ une fonction qui associe un poids positif $w(c)$ à chaque contrainte $c \in \mathcal{C}$, on note $U_{\mathcal{C}}(a)$ l'ensemble des contraintes violées par a (i.e. $c(a) = 1$). Le problème de satisfaction maximum de contraintes (MWCSP) consiste à trouver une affectation complète a minimisant la somme des poids des contraintes violées par a:

$$f_{\mathcal{C}}(w, a) = \sum_{c \in U_{\mathcal{C}}(a)} w(c)$$

Une définition de la satisfaction partielle de contraintes est également donnée dans la section 1.1 de ce mémoire. Étant donné un CSP irréalisable P, une fonction de distance d et une borne β, le MPCSP consiste à trouver un CSP réalisable P' tel que la distance $d(P, P')$ entre P et P' soit minimale (et inférieure à β). Par ailleurs, plusieurs manières de relaxer P ont été données. Parmi celles-ci, P peut être relaxé en permettant à certaines de ses variables de ne pas être instanciées. Il a été montré que retirer une variable x permettait d'étendre l'ensemble des affectations autorisées des variables $\mathcal{X}(c)$ d'une contrainte c agissant sur x. En particulier, retirer une variable correspond à retirer du CN toutes les contraintes unaires et binaires agissant sur cette variables. En considérant P comme un CSP incohérent dont les affectations doivent être complètes, la distance $d(P, P')$ à minimiser est alors le nombre de variables devant être retirées de P (i.e. autorisées à ne pas être instanciées) afin d'avoir une affectation partielle légale pour P'. Cette définition du MPCSP peut également être étendue pour tenir compte des pondérations sur les variables:

Définition Soit un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ et $w : \mathcal{X} \rightarrow \mathbb{R}^+$ une fonction qui associe un poids positif $w(x)$ à chaque variable $x \in \mathcal{X}$, on note $U_{\mathcal{X}}(a)$ l'ensemble des variables n'étant pas instanciées dans a. Le problème de satisfaction partielle maximum de contraintes (MPWCSP) consiste à trouver une affectation légale a minimisant la somme des poids des variables n'étant pas instanciées dans a:

$$f_{\mathcal{X}}(w, a) = \sum_{x \in U_{\mathcal{X}}(a)} w(x)$$

En somme, le MWCSP correspond à chercher une affectation complète telle que la somme des poids des contraintes violées est minimum, alors que le MPWCSP consiste à chercher une affectation partielle légale minimisant la somme des poids des variables non-instanciées. Le MWCSP et le MPWCSP sont deux problèmes NP-difficiles. On peut ainsi transformer en MWCSP ou en MPWCSP un CSP, qui est reconnu NP-complet, en donnant un poids $w(c) > 0$ à chaque contrainte $c \in \mathcal{C}$ et un poids $w(x) > 0$ à chaque variable $x \in \mathcal{X}$. Le CSP est alors satisfaisable s'il existe, pour le MWCSP correspondant, une affectation a telle que $f_{\mathcal{C}}(w, a) = 0$. De même, un CSP est satisfaisable s'il existe, pour le MPWCSP correspondant, une affectation légale telle que $f_{\mathcal{X}}(w, a) = 0$.

Une des raisons pour laquelle des poids ont été ajoutés aux variables et aux contraintes du CN est que ces poids permettent de modifier dynamiquement le CN, et conséquemment les problèmes de satisfaction correspondants. Ainsi, donner un poids de 0 à une variable x du CN équivaut à retirer cette variable dans le cas du MPWCSP, puisque la fonction objectif $f_{\mathcal{X}}$ n'augmentera pas si x n'est pas instanciée dans une affectation quelconque. De même, si le poids d'une contrainte vaut 0, il est alors possible de violer cette contrainte dans une affectation a du MWCSP sans augmenter $f_{\mathcal{C}}(w, a)$. Par ailleurs, il est également possible de forcer la satisfaction d'une contrainte ou l'instanciation d'une variable en augmentant son poids,

c'est-à-dire en rendant "dure" cette variable ou cette contrainte. Par exemple, si on donne un poids de $|\mathcal{X}|$ à une variable $x \in \mathcal{X}$ et un poids de 1 aux autres variables, on s'assure que x sera instanciée dans toute affectation optimale au MPWCSP. On peut aussi forcer la satisfaction d'une contrainte $c \in \mathcal{C}$ dans toute affectation optimale au MWCSP, en donnant à c un poids de $|\mathcal{C}|$, et un poids de 1 aux autres contraintes de $\mathcal{C}$.

Il est, par ailleurs, important de remarquer que, pour un CN donné, le MWCSP et le MPWCSP sont des problèmes équivalents aux problèmes de recouvrement de poids minimum des IIS (MWIC) de contraintes et de variables du CN (*voir section 2.1.5*). Ainsi, dans le cas du MWCSP, toute affectation complète a doit violer une contrainte de chaque IIS de contraintes. L'ensemble $U_{\mathcal{C}}(a)$ des contraintes violées par a est donc un HS des IIS de contraintes du CN. En plus, si a est optimale, $U_{\mathcal{C}}(a)$ correspond à un HS de poids minimum. De même, dans le cas du PWCSP, toute affectation partielle doit avoir une variable de chaque IIS de variables non-instanciée. L'ensemble $U_{\mathcal{X}}(a)$ des variables non-instanciées est donc un HS des IIS de variables du CN. Enfin, si a est optimale, $U_{\mathcal{X}}(a)$ correspond à un HS de poids minimum.

Il est nécessaire d'illustrer ces notions sur un exemple. La figure 3.1 représente un graphe de six sommets et sept arêtes, et dont le nombre chromatique est 3. Le problème de 2-coloriage de ce graphe correspond donc à un CSP irréalisable possédant deux IIS de contraintes: $\{c_1, c_2, c_4\}$ qui est minimum, et $\{c_3, c_4, c_5, c_6, c_7\}$. Ce CSP contient également deux IIS de variables: $\{x_1, x_2, x_6\}$ qui est minimum, et $\{x_2, x_3, x_4, x_5, x_6\}$. En supposant que toutes les contraintes et les variables ont le même poids, l'ensemble $U_{\mathcal{C}}(a)$ pour une affectation a optimale au MWCSP est un HS de cardinalité minimum des IIS de contraintes. Ainsi, comme c_4 est la seule contrainte faisant partie des deux IIS de contraintes, une affectation optimale viole uniquement c_4, par exemple $a = (1, 2, 1, 2, 1, 2)$. De la même manière, l'ensemble

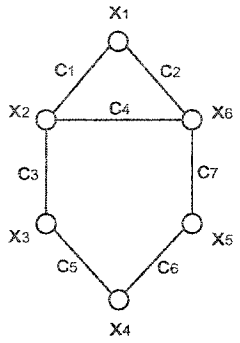

Figure 3.1 Une instance de k-coloriage incohérente pour $k \leq 2$

$U_{\mathcal{X}}(a)$ pour une affectation a optimale au MPWCSP est un HS de cardinalité minimale des IIS de variables. Puisque les seules variables faisant partie des deux IIS de variables sont x_2 et x_6, une affectation optimale au MPWCSP doit avoir x_2 ou x_6 désinstanciée, par exemple $a = (1, -, 1, 2, 1, 2)$ ("-" signifie que la variable correspondante n'est pas instanciée). Par ailleurs, en changeant le poids des contraintes et des variables du CN, on obtient différentes affectations optimales au MWCSP et au MPWCSP. Ainsi, si on donne un poids de 1 à c_1 et c_3, et un poids de $|\mathcal{C}| = 7$ aux autres contraintes, une affectation optimale au MWCSP viole nécessairement c_1 et c_3 puisque ces contraintes forment un HS de poids minimum 2 des IIS de contraintes. De plus, si on donne un poids de $|\mathcal{X}| = 6$ à x_2, et un poids de 1 aux autres, une affectation optimale au MPWCSP doit avoir x_6 non-instanciée, puisque cette variable correspond à un HS de poids minimum 1 des IIS de variables.

3.2 Algorithmes de détection

Les algorithmes présentés dans cette section peuvent être utilisés pour trouver aussi bien des IIS de contraintes que des IIS de variables. Afin d'alléger leur description, une seule version sera donnée pour chaque algorithme, pouvant s'appliquer à la détection d'IIS de contraintes ou de variables. Ainsi, si le but est de trouver un IIS

de contraintes, $\mathcal{E}$ correspond à l'ensemble de contraintes $\mathcal{C}$, w est une fonction qui pondère les contraintes de $\mathcal{C}$, f est la fonction objectif du MWCSP $f_{\mathcal{C}}$, $U(a)$ est l'ensemble des contraintes violées par une affectation complète a, et *procMWIC* est une procédure qui retourne une affectation complète minimisant $f_{\mathcal{C}}$. Par contre, si la tâche est de trouver un IIS de variables, alors $\mathcal{E}$ correspond à l'ensemble de variables $\mathcal{X}$, w est une fonction qui pondère les variables de $\mathcal{X}$, f est la fonction objectif du MPWSCP $f_{\mathcal{X}}$, $U(a)$ est l'ensemble des variables n'étant pas instanciées dans une affectation partielle légale a, et *procMWIC* est une procédure qui retourne une affectation partielle légale minimisant $f_{\mathcal{X}}$.

Puisque *procMWIC* résoud de manière optimale le MWCSP (dans le cas de la détection d'IIS de contraintes) et le MPWCSP (dans le cas de la détection d'IIS de variables), deux problèmes NP-difficiles, la complexité en temps de calcul des algorithmes de détection est presque entièrement attribuable à l'emploi de cette procédure[1]. De manière générale, cette procédure est appelée une seule fois par itération d'un algorithme de détection, correspondant à une modification des poids des contraintes ou variables du CN. Ainsi, pour chaque algorithme de détection, une indication de la complexité en temps de calcul sera donnée sous la forme du nombre d'itérations nécessaire, dans le pire cas, pour trouver un IIS. Lorsque cela est possible, cette valeur sera exprimée en terme de $|\mathcal{E}|$ (i.e. $|\mathcal{X}|$ ou $|\mathcal{C}|$) ou de $|K|$.

3.2.1 Algorithme de retrait

L'algorithme de *retrait* est probablement le plus simple des algorithmes de détection d'IIS. Des approches similaires ont déjà été proposées pour la programmation linéaire (Chinneck, 1997 (1); Chinneck, 1997 (2)) et le problème de coloriage de

[1] À l'exception de l'algorithme de *hitting set* qui doit également résoudre le problème NP-difficile du *hitting set* minimum

graphe (Herrmann et Hertz, 2002). Cet algorithme prend en paramètre un CN incohérent P et retourne un IIS de contraintes ou de variables K. Tout d'abord, le poids de chaque contrainte ou variable de $\mathcal{E}$ est initialisé à 1. Ensuite, ces contraintes ou variables sont temporairement retirées selon un ordre quelconque, en leur donnant un poids de 0. Après chaque retrait, une affectation optimale a est obtenue avec *procMWIC*. Si $f(w, a) = 0$, P est alors devenu cohérent, et la dernière contrainte ou variable retirée est remise dans P en lui donnant un poids de 1. Lorsque toutes les contraintes ou les variables ont été testées, celles dont le poids vaut 1 forment un IIS.

Algorithme 5 Algorithme de retrait

Entrée: Un CN incohérent P;
Sortie : Un IIS de contraintes ou de variables K.

Initialisation
pour tout $e \in \mathcal{E}$ **faire**
  $w(e) \leftarrow 1$;
fin pour
$T \leftarrow \mathcal{E}$;

Construction
tant que $T \neq \emptyset$ **faire**
  Choisir un élément $e \in T$;
  $T \leftarrow T \setminus \{e\}$;
  $w(e) \leftarrow 0$;
  $a \leftarrow$ procMWIC(P, w);
  si $f(w, a) = 0$ **alors**
    $w(e) \leftarrow 1$;
  fin si
fin tant que

Extraction
$K \leftarrow \{e \mid w(e) = 1\}$;

Propriété 3.2.1 *Soit P un CN incohérent. L'algorithme de retrait produit, en un nombre fini d'itérations, un IIS de contraintes ou de variables K de P.*

Preuve On remarque tout d'abord que chaque retrait rendant P cohérent est suivi d'une ré-insertion rendant P à nouveau incohérent. L'ensemble K est donc incohérent. De plus, soit e_i n'importe quelle contrainte ou variable de K retirée à l'itération i, et soit K_i l'ensemble des contraintes ou variables ayant un poids de 1 a l'itération i. On sait que $K_i \setminus \{e_i\}$ est cohérent. Par ailleurs, comme $K \subseteq K_i$, $K \setminus \{e_i\}$ est également cohérent. L'ensemble K est donc un IIS.

Afin d'illustrer l'algorithme de retrait, prenons le problème de satisfaction correspondant au 2-coloriage du graphe de la figure 3.1. Supposons que le but soit d'obtenir un IIS de contraintes et que les contraintes soient retirées selon l'ordre croissant de leur indice. Ainsi, c_1 est d'abord retirée en lui donnant un poids de 0. On remarque que le graphe résultant n'est toujours pas 2-coloriable. De plus, comme ce retrait a détruit un des deux IIS (i.e. $\{c_1, c_2, c_4\}$), *procMWIC* obtient une affectation optimale a_1 de coût $f(w, a_1) = 1$, et c_1 n'est pas ré-insérée. La contrainte c_2 est ensuite retirée. Comme c_2 ne fait pas partie de l'IIS restant, on obtient encore une affectation optimale a_2 de coût $f(w, a_2) = 1$, et c_2 n'est pas ré-insérée. Toutes les contraintes de poids 1 font alors partie de l'IIS restant. Ainsi, en retirant c_3, le graphe devient 2-coloriable (i.e. $f(w, a_3) = 0$), et cette contrainte est alors ré-insérée en lui redonnant un poids de 1. La même chose se produit pour les contraintes c_4, c_5, c_6 et c_7, de sorte que l'algorithme produit l'IIS $K = \{c_3, c_4, c_5, c_6, c_7\}$. Il est possible d'obtenir un IIS différent en modifiant l'ordre de retrait des contraintes. Ainsi, en considérant l'ordre inverse, c_7 est d'abord retirée et un IIS différent est détruit. Conséquemment K sera nécessairement formé par $\{c_1, c_2, c_4\}$, soit l'IIS minimum. Dans le cas de la détection d'IIS de variables, en retirant les variables selon l'ordre croissant de leur indice, x_1 est d'abord retirée. Comme le graphe résultant n'est toujours pas 2-coloriable, $f(w, a_1) = 1$ et x_1 n'est pas ré-insérée. Cependant, lorsque x_2 est retirée, $f(w, a_2) = 0$ et cette variable est ré-insérée en lui donnant un poids de 1. La même chose se produit pour les

autres variables, qui font toutes partie du seul IIS restant, et l'algorithme retourne $K = \{x_2, x_3, x_4, x_5, x_6\}$. Encore une fois, un IIS différent peut être obtenu en modifiant l'ordre de retrait. Ainsi, si on retire x_3, x_4 ou x_5 avant x_1, l'IIS obtenu sera $\{x_1, x_2, x_6\}$.

Puisque toutes les contraintes ou variables de $|\mathcal{E}|$ sont testées, l'algorithme de retrait met, dans tous les cas, $|\mathcal{E}|$ itérations pour obtenir K. On remarque, par ailleurs, que l'IIS obtenu par cet algorithme est celui ayant une première contrainte ou variable retirée en dernier, puisque les autres IIS auront alors été détruits. De plus, on constate que tous les IIS de ce problème peuvent être obtenus en variant l'ordre de retrait, et que des ordres de retrait différents peuvent produire le même IIS. Cependant, comme le nombre d'ordres de retrait différents (i.e. $|\mathcal{E}|!$) est supérieur au nombre de sous-ensembles de $\mathcal{E}$ (i.e. $2^{|\mathcal{E}|}$), il est nécessaire d'avoir des heuristiques pour déterminer un ordre de retrait pouvant produire un IIS minimum ou ayant moins de contraintes ou de variables. Une telle heuristique sera présentée à la fin de ce chapitre.

3.2.2 Algorithme d'insertion

Alors que l'algorithme de retrait obtient un IIS en retirant des contraintes ou des variables, l'algorithme d'*insertion* en produit un en les rajoutant à K. Comme le précédent, cet algorithme prend en paramètre un CN incohérent P et retourne un IIS K. Au départ, le poids des contraintes ou variables de $\mathcal{E}$ est initialisé à 1. Ensuite, pour chaque itération i, *procMWIC* retourne une affectation optimale a_i minimisant f. Cette affectation a un ensemble $U(a_i)$ de contraintes violées ou de variables n'étant pas instanciées. Le poids d'une contrainte ou variable de cet ensemble est fixé à $|\mathcal{E}|$, et celui des autres fixé à 0. Ainsi, puisque $U(a_i)$ contient une contrainte ou variable de chaque IIS, tous les IIS contenant la contrainte ou

la variable dont le poids a été fixé à $|\mathcal{E}|$ seront conservés, et les autres détruits. Ce processus est répété jusqu'à ce que les contraintes ou variables de poids $|\mathcal{E}|$ forment un ensemble K incohérent. Alors, *procMWIC* obtient une affectation a tel que $U(a)$ contient une contrainte ou variable de cet ensemble (i.e. $f(w,a) \geq |\mathcal{E}|$). Finalement, l'algorithme retourne l'ensemble K qui forme un IIS.

Algorithme 6 Algorithme d'insertion

Entrée: Un CN incohérent P;
Sortie : Un IIS de contraintes ou de variables K.

Initialisation
pour tout $e \in \mathcal{E}$ **faire**
  $w(e) \leftarrow 1$;
fin pour

Construction
répéter
  $a \leftarrow$ procMWIC(P, w);
  si $f(w,a) < |\mathcal{E}|$ **alors**
    Choisir un élément $e \in U(a)$ tel que $w(e) = 1$;
    $w(e) \leftarrow |\mathcal{E}|$;
    pour tout $e' \in U(a)$, $e' \neq e$ **faire**
      $w(e') \leftarrow 0$;
    fin pour
  fin si
juqu'à ce que $f(w,a) \geq |\mathcal{E}|$

Extraction
$K \leftarrow \{e \mid w(e) = |\mathcal{E}|\}$;

Propriété 3.2.2 *Soit P un CN incohérent. L'algorithme d'insertion produit, en un nombre fini d'itérations, un IIS de contraintes ou de variables K de P.*

Preuve On constate, tout d'abord, qu'une contrainte ou variable d'au moins un IIS est conservée à chaque itération. Conséquemment, K est un ensemble incohérent. Par ailleurs, soit a_i l'affectation optimale obtenue à l'itération i, et $e_i \in U(a_i)$ n'importe quelle contrainte ou variable de K conservée à cette itération.

On sait que $\mathcal{E} \backslash U(a_i)$ est un ensemble cohérent. De plus, comme $K \backslash \{e_i\} \subseteq \mathcal{E} \backslash U(a_i)$, l'ensemble $K \setminus \{e_i\}$ est aussi cohérent. L'ensemble K est donc un IIS.

Illustrons l'algorithme d'insertion sur le problème irréalisable de 2-coloriage du graphe de la figure 3.1. Dans le cas de la détection d'IIS de contraintes, *procMWIC* retourne d'abord une affectation a_1 tel que $U(a_1) = \{c_4\}$. Puisque c_4 est la seule contrainte violée, son poids est fixé à $|\mathcal{C}| = 7$. L'affectation suivante a_2 doit alors satisfaire c_4 et violer une contrainte de chaque IIS, par exemple $U(a_2) = \{c_1, c_3\}$. Supposons que l'on fixe le poids de c_1 à 0 et celui de c_3 à 7, il ne reste qu'un seul IIS: $\{c_3, c_4, c_5, c_6, c_7\}$. Donc, les affectations a_3, a_4 et a_5 vont fixer le poids des contraintes c_5, c_6, et c_7 à 7. Comme les contraintes de poids 7 forment un ensemble incohérent, l'affectation a_6 aura alors un coût $f(w, a_6) = 7$, et cet IIS est détecté. Une fois de plus, le choix de la contrainte de chaque $U(a_i)$ à laquelle fixer un poids de $|\mathcal{C}|$ détermine quel IIS est obtenu. Ainsi, si on avait conservé c_1, au lieu de c_3, à l'itération 2, l'IIS contenant c_3 aurait été détruit et celui formé de c_1, c_2 et c_4, qui est minimum, aurait été obtenu. Dans le cas de la détection d'IIS de variables, la première affectation a_1 donne un ensemble $U(a_1)$ contenant x_2 ou x_6, par exemple, x_6. Le poids de x_6 est alors fixé à $|\mathcal{X}| = 6$. Ensuite, a_2 donne $U(a_2) = \{x_2\}$, et on fixe le poids de x_2 à 6. L'affectation a_3 doit alors produire un ensemble $U(a_3)$ contenant x_1 et une autre variable de l'ensemble $\{x_3, x_4, x_5\}$, par exemple x_3. Supposons que l'on fixe le poids de x_3 à 6 et celui de x_1 à 0, le poids de x_4 et x_5 sera ensuite fixé à 6, et l'IIS obtenu sera $\{x_2, x_3, x_4, x_5, x_6\}$. Si, par contre, on avait fixé le poids de x_3 à 0 et celui de x_1 à 6, l'algorithme aurait obtenu l'IIS minimum $\{x_1, x_2, x_6\}$.

Comme le poids d'une contrainte ou variable de K est fixé à $|\mathcal{E}|$ à chaque itération, et que l'algorithme se termine lorsque K est incohérent, le nombre d'itérations

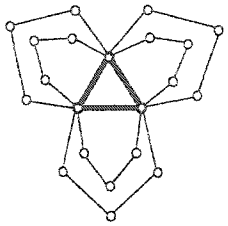

Figure 3.2 Une instance ayant un IIS minimum de contraintes pour $k = 2$

mis par l'algorithme d'insertion pour obtenir un IIS est donc $|K| + 1$ [2]. Ainsi, puisque $|K| \leq |\mathcal{E}|$, l'algorithme d'insertion est plus rapide que celui de retrait, particulièrement pour des CN contenant un petit IIS. Par exemple, l'algorithme de retrait mettra 1000 itérations pour trouver un IIS de 10 variables dans un CN contenant 1000 variables, alors que l'algorithme d'insertion n'en mettra que 11. Par contre, contrairement à l'algorithme de retrait, l'algorithme d'insertion ne permet pas de détecter tous les IIS d'un CN incohérent. Considérons, par exemple, la détection d'un IIS de contraintes dans le problème irréalisable de 2-coloriage du graphe de la figure 3.2. La première affectation doit violer les trois contraintes du centre[3]. Ces contraintes forment le seul IIS minimum. Comme le poids de deux de ces contraintes doit être fixé à 0, cet IIS sera alors détruit. Il est donc impossible d'obtenir cet IIS minimum à l'aide de l'algorithme d'insertion.

Finalement, il n'est pas nécessaire de fixer à $|\mathcal{E}|$ le poids de la contrainte ou variable conservée à chaque itération. Cette valeur est utilisée parce qu'elle garantit qu'une affectation satisfaisant toutes les contraintes dures ou instanciant toutes les variables dures ait un coût inférieur à toute affectation violant une de ces contraintes ou pour laquelle une de ces variables n'est pas instanciée. Il suffit, en réalité, de s'assurer que cette valeur soit supérieure au plus grand nombre de contraintes ou variables de poids 1, dans chaque ensemble $U(a_i)$. Ainsi, l'exemple de la figure 3.1

[2]Il faut une itération supplémentaire pour détecter l'incohérence.
[3]En gras dans la figure.

contient seulement deux IIS de contraintes et de variables, tel que $U(a_i) \leq 2$. On peut alors utiliser 3 au lieu de $|\mathcal{C}| = 7$ et $|\mathcal{X}| = 6$

3.2.3 Algorithme de *hitting set*

Contrairement aux deux algorithmes de détection précédents, l'algorithme de *hitting set* permet d'obtenir des IIS de cardinalité minimum. Cet algorithme se base sur le principe que chaque affectation a_i, retournée par *procMWIC* à l'itération i, produit un ensemble $U(a_i)$ contenant une contrainte ou une variable de chaque IIS. Un IIS est donc un HS de la collection $\mathcal{U} = \{U(a_1), \ldots, U(a_{|\mathcal{U}|})\}$. Par ailleurs, comme cet algorithme utilise une procédure *procMHS*($\mathcal{U}$) qui retourne un HS minimum de $\mathcal{U}$, l'IIS obtenu est lui aussi minimum.

Algorithme 7 Algorithme de *hitting set*

Entrée: Un CN incohérent P;
Sortie : Un IIS de contraintes ou de variables K.

Initialisation
$\mathcal{U} \leftarrow \emptyset$;

Construction
répéter
  $K \leftarrow$ procMHS($\mathcal{U}$);
  pour tout $e \in \mathcal{E}$ **faire**
    si $e \in K$ **alors**
      $w(e) \leftarrow |\mathcal{E}|$;
    sinon
      $w(e) \leftarrow 1$;
    fin si
  fin pour
  $a \leftarrow$ procMWIC(P, w);
  si $f(w, a) < |\mathcal{E}|$ **alors**
    $\mathcal{U} \leftarrow \mathcal{U} \cup \{U(a)\}$;
  fin si
juqu'à ce que $f(w, a) \geq |\mathcal{E}|$

À chaque itération i, l'algorithme de *hitting set* obtient un ensemble K_i qui est un HS d'une collection $\mathcal{U}$ initialement vide. Le poids des contraintes ou des variables de $\mathcal{E}$ est ensuite modifié de manière à ce que celles faisant partie de K_i reçoivent un poids de $|\mathcal{E}|$, et les autres un poids de 1. Une affectation optimale a_i est ensuite obtenue de *procMWIC* et l'ensemble $U(a_i)$ est ajouté à $\mathcal{U}$. Finalement, lorsqu'une affectation optimale a_i de coût $f(w, a) \geq |\mathcal{E}|$ est trouvée, K_i est un ensemble incohérent et l'algorithme se termine. Cet ensemble est alors un IIS minimum de contraintes ou de variables.

Propriété 3.2.3 *Soit P un CN incohérent. L'algorithme de* hitting set *produit, en un nombre fini d'itérations, un IIS de contraintes ou de variables K de P, si* procMHS *retourne des HS minimaux au sens de l'inclusion. Par ailleurs, si* procMHS *retourne des HS de cardinalité minimale, K est un IIS minimum.*

Preuve Soit K_i le HS obtenu à l'itération i, et a_i l'affectation optimale de cette itération. Si $f(w, a_i) \geq |\mathcal{E}|$, l'algorithme se termine, sinon on sait que $U(a_i) \cap K_i = \emptyset$. Soit K_j le HS obtenu à une itération $j > i$, K_j doit contenir une contrainte ou une variable de chaque ensemble $\{U(a_1), \ldots, U(a_{j-1})\}$, en particulier de $U(a_i)$. L'ensemble K_j contient donc une contrainte ou variable de $U(a_i)$ qui n'était pas dans K_i. Conséquemment $K_i \neq K_j$. De plus, comme le nombre de HS K possibles n'est pas supérieur au nombre de sous-ensembles de $\mathcal{E}$, l'algorithme de *hitting set* doit se terminer en un nombre fini d'itérations. Par ailleurs, comme l'algorithme ne se termine que lorsque K est un ensemble incohérent, et puisque *procMHS* retourne des HS minimaux au sens de l'inclusion, K est donc un IIS. Finalement, si *procMHS* retourne des HS de cardinalité minimale, K est nécessairement un IIS minimum.

Afin d'illustrer l'algorithme de *hitting set*, considérons, une fois de plus, le problème de 2-coloriage du graphe de la figure 3.1. Commençons d'abord par un mau-

vais scénario, où le seul IIS minimum de contraintes du problème est seulement trouvé après 7 itérations. Au départ, comme $\mathcal{U}$ est vide, $K_1 = \emptyset$ et le poids de toutes les contraintes de $\mathcal{E}$ est fixé à 1. Ensuite, la première affectation a_1 obtenue par *procMWIC* produit l'ensemble $U(a_1) = \{c_4\}$ qui est ajouté à $\mathcal{U}$. Conséquemment, *procMHS* obtient un HS $K_2 = \{c_4\}$, tel que le poids de c_4 est fixé à $\mathcal{C} = 7$ et celui des autres contraintes à 1. L'ensemble $U(a_2)$ doit alors contenir une contrainte de l'ensemble $\{c_1, c_2\}$ et une autre de $\{c_3, c_5, c_6, c_7\}$, par exemple $U(a_2) = \{c_1, c_3\}$. Cet ensemble est ajouté à $\mathcal{U}$ qui devient $\mathcal{U} = \{\{c_4\}, \{c_1, c_3\}\}$. Le HS suivant doit alors contenir c_4 ainsi qu'une contrainte de $\{c_1, c_3\}$, par exemple c_1, ce qui donne $K_3 = \{c_1, c_4\}$, tel que $U(a_3)$ contient c_2 et une autre contrainte parmi $\{c_3, c_5, c_6, c_7\}$, par exemple, c_3. Cet ensemble est alors ajouté à $\mathcal{U}$ qui devient $\mathcal{U} = \{\{c_4\}, \{c_1, c_3\}, \{c_2, c_3\}\}$, et le prochain HS peut, par exemple, être $K_4 = \{c_3, c_4\}$. L'ensemble $U(a_4)$ doit alors contenir une contrainte de $\{c_1, c_2\}$ et une autre de $\{c_5, c_6, c_7\}$, supposons c_1 et c_5, tel que $\mathcal{U} = \{\{c_4\}, \{c_1, c_3\}, \{c_2, c_3\}, \{c_1, c_5\}\}$. Le prochain HS obtenu par *procMHS* peut alors être $K_5 = \{c_1, c_3, c_4\}$, tel que $U(a_5)$ doit contenir c_2 et une autre contrainte de $\{c_5, c_6, c_7\}$, par exemple, c_5. Cet ensemble est ensuite ajouté à $\mathcal{U}$ pour donner l'ensemble $\mathcal{U} = \{\{c_4\}, \{c_1, c_3\}, \{c_2, c_3\}, \{c_1, c_5\}, \{c_2, c_5\}\}$, contenant un seul HS minimum $K_6 = \{c_3, c_4, c_5\}$. L'ensemble $U(a_6)$ contient alors une contrainte de $\{c_1, c_2\}$ et une autre de $\{c_6, c_7\}$, par exemple, c_1 et c_6, et on obtient $\mathcal{U} = \{\{c_4\}, \{c_1, c_3\}, \{c_2, c_3\}, \{c_1, c_5\}, \{c_2, c_5\}, \{c_1, c_6\}\}$. Finalement, cet ensemble ne contient qu'un seul HS minimum $K_7 = \{c_1, c_2, c_4\}$ qui est l'IIS minimum de contraintes. Il est à noter que la détection de cet IIS minimum aurait pu être plus rapide. Ainsi, à l'itération 5, *procMHS* aurait pu retourner le HS $K_5 = \{c_1, c_2, c_4\}$ formant l'IIS minimum.

Dans le cas de la détection d'IIS de variables, $U(a_1)$ doit contenir x_2 ou x_6, par exemple $U(a_1) = \{x_2\}$. Cet ensemble est ajouté à $\mathcal{U}$ pour donner $\mathcal{U} = \{\{x_2\}\}$ qui

possède un HS minimum $K_2 = \{x_2\}$. Le poids de x_2 est alors fixé à $\mathcal{X} = 6$ et celui des autres variables à 1, tel que a_2 donne l'ensemble $U(a_2) = \{x_6\}$. L'ensemble $\mathcal{U}$ devient ensuite $\mathcal{U} = \{\{x_2\}, \{x_6\}\}$ dont le HS minimum est $K_3 = \{x_2, x_6\}$. On obtient alors un ensemble $U(a_3)$ contenant x_1 et autre variable parmi $\{x_3, x_4, x_5\}$. par exemple x_3, et $\mathcal{U}$ devient alors $\mathcal{U} = \{\{x_2\}, \{x_6\}, \{x_1, x_3\}\}$. Supposons que *procMHS* retourne $K_4 = \{x_2, x_3, x_6\}$, $U(a_4)$ peut alors être $\{x_1, x_4\}$ ou $\{x_1, x_5\}$. Qu'il s'agisse de l'un ou de l'autre, le HS suivant sera $K_5 = \{x_1, x_2, x_6\}$ qui est l'IIS minimum. Finalement, la détection aurait également pu être plus courte. Ainsi, à l'itération 4, le HS minimum obtenu par *procMHS* aurait pu être $K_4 = \{x_1, x_2, x_6\}$, l'IIS minimum.

Alors que l'algorithme de *hitting set* obtient un IIS minimum, cette détection prend un nombre exponentiel d'itérations. À chaque itération i, *procMHS* produit un HS K_i différent de celui produit aux itérations précédentes. Un IIS minimum K sera donc obtenu dans un nombre fini d'itérations. Cependant, comme le nombre de sous-ensembles possibles de $\mathcal{E}$ contenant $|K|$ éléments est donné par

$$C(|\mathcal{E}|, |K|) = \frac{|\mathcal{E}|!}{|\mathcal{E}|! \ (|\mathcal{E}| - |K|)!}$$

un nombre exponentiel de HS peut être produit avant d'obtenir K.

Finalement, si *procMHS* obtient des HS de cardinalité minimum, on peut montrer que la taille de l'ensemble K_i des contraintes ou des variables de poids $|\mathcal{E}|$, à n'importe quelle itération i, est une borne inférieure sur la taille d'un IIS minimum K de P (i.e. $|K_i| \leq |K|$). Ainsi, même si l'algorithme de *hitting set* ne parvient pas à obtenir un IIS minimum K dans un temps raisonnable, on peut l'arrêter à une itération i quelconque, et utiliser $|K_i|$ comme borne sur la taille de K.

3.3 Techniques complémentaires de détection

Cette section présente des variantes des algorithmes décrits précédemment, ainsi que des techniques permettant de faciliter la détection d'un IIS. Ces variantes et techniques ont été développées dans le cadre de ce mémoire.

3.3.1 Algorithme hybride

L'algorithme de retrait, présenté au début de ce chapitre, permet de détecter un IIS en retirant une à une les contraintes ou les variables de $\mathcal{E}$. Cet algorithme obtient donc un IIS après $|\mathcal{E}|$ itérations, peu importe la taille de cet IIS. De son côté, l'algorithme d'insertion, qui fixe les poids des contraintes ou variables à $|\mathcal{E}|$, obtient un IIS K en $|K|$ itérations. L'algorithme présenté dans cette section intègre certains principes de l'algorithme d'insertion à l'algorithme de retrait, afin de permettre à ce dernier d'obtenir un IIS plus rapidement.

Comme l'algorithme de retrait, l'algorithme *hybride* retire à chaque itération i une contrainte ou une variable $e \in \mathcal{E}$ en fixant son poids à 0. Si $f(w, a_i) > 0$, P est toujours incohérent et e n'est pas ré-insérée. Par contre, si $f(w, a_i) = 0$, e fait nécessairement partie de l'intersection des IIS de P. Contrairement à l'algorithme de retrait, l'algorithme *hybride* ré-insère e en fixant son poids à $|\mathcal{E}|$. Ainsi, lorsque *procMWIC* retourne une affectation a_i de coût $f(w, a_i) \geq |\mathcal{E}|$, l'ensemble incohérent formé des contraintes ou variables ayant un poids de $\mathcal{E}$ est un IIS. En conséquence, l'algorithme *hybride* peut obtenir un IIS K sans avoir à retirer toutes les contraintes ou les variables. Finalement, sachant qu'à n'importe quelle itération, les contraintes ou variables de poids $\mathcal{E}$ font partie de l'IIS K, on peut utiliser une heuristique, comme celle présentée à la fin de ce chapitre, pour choisir la prochaine contrainte ou variable à retirer afin d'avoir K le plus petit possible.

Algorithme 8 Algorithme hybride

Entrée: Un CN incohérent P;
Sortie : Un IIS de contraintes ou de variables K.

Initialisation
pour tout $e \in \mathcal{E}$ **faire**
  $w(e) \leftarrow 1$;
fin pour

Construction
répéter
  Choisir un élément $e \in \mathcal{E}$ tel que $w(e) = 1$;
  $w(e) \leftarrow 0$;
  $a \leftarrow \text{procMWIC}(P, w)$;
  si $f(w, a) = 0$ **alors**
    $w(e) \leftarrow |\mathcal{E}|$;
  fin si
juqu'à ce que $f(w, a) \geq |\mathcal{E}|$

Extraction
$K \leftarrow \{e \mid w(e) = |\mathcal{E}|\}$;

Propriété 3.3.1 *Soit P un CN incohérent. L'algorithme hybride produit, en un nombre fini d'itérations, un IIS de contraintes ou de variables K de P.*

Preuve On remarque tout d'abord que chaque retrait rendant P cohérent est suivi d'une ré-insertion rendant P à nouveau incohérent. L'ensemble K est donc incohérent. De plus, soit e_i n'importe quelle contrainte ou variable de K retirée à l'itération i, et soit K_i l'ensemble des contraintes ou variables ayant un poids supérieur à 0 à l'itération i. On sait que $K_i \setminus \{e_i\}$ est cohérent. Par ailleurs, comme $K \subseteq K_i$, $K \setminus \{e_i\}$ est également cohérent. L'ensemble K est donc un IIS.

L'exemple suivant illustre l'algorithme hybride dans le cas de la détection d'IIS de contraintes sur le problème de 2-coloriage du graphe de la figure 3.1. Si les contraintes sont retirées selon l'ordre croissant de leur indice, l'algorithme hybride mettra sept itérations pour détecter l'IIS $K = \{c_3, c_4, c_5, c_5, c_7\}$, exactement comme

l'algorithme de retrait. Par contre, supposons que les contraintes soient retirées selon l'ordre suivant: $\{c_4, c_3, c_1, c_2, c_5, c_6, c_7\}$. Lorsque c_4 est retirée, P devient cohérent. Comme cette contrainte fait nécessairement partie de K, on la ré-insère en fixant son poids à $\mathcal{C} = 7$. Ensuite, c_3 est retirée, et puisque le CN ne devient pas cohérent, cette contrainte n'est pas ré-insérée. Les retraits de c_1 et c_2 rendent ensuite le CN cohérent, et le poids de ces contraintes est fixé à 7. L'affectation a_5 a alors un coût $f(w, a_5) = 7$ et l'algorithme retourne l'ensemble $K = \{c_1, c_2, c_4\}$ qui est l'IIS minimum de contraintes. On remarque que la détection cet IIS a nécessité cinq itérations au lieu de sept.

Comme le montre l'exemple précédent, l'algorithme met, dans le pire cas, $|\mathcal{E}|$ itérations à détecter un IIS. Cependant, après avoir fait plusieurs retraits, un seul IIS devrait rester dans le CN. Il suffit alors de retirer chaque contrainte ou variable de cet IIS pour le détecter. Cette technique est surtout utile lorsque l'IIS restant contient une petite proportion des contraintes ou variables du CN.

3.3.2 Algorithme de retour-arrière

Les algorithmes présentés jusqu'à maintenant se concentrent sur la détection d'un seul IIS et se terminent lorsque cet IIS est obtenu. L'algorithme suivant utilise une stratégie de retour-arrière pour obtenir le plus petit IIS possible.

On se rappelle que l'algorithme d'insertion obtient, à chaque itération i, une affectation a_i telle que $U(a_i)$ contient au moins une contrainte ou une variable de chaque IIS du CN (i.e $U(a_i)$ est un HS des IIS du CN). Le poids d'une contrainte ou d'une variable $e \in U(a_i)$ est ensuite fixé à $|\mathcal{E}|$ et celui des autres à 0. Parmi les IIS restant à cette itération, tous ceux qui ne contiennent pas e sont alors détruits. Le choix de quelle contrainte ou variable conserver à chaque itération détermine donc l'IIS

obtenu K. On peut, par ailleurs, imaginer un arbre pour lequel chaque niveau, correspondant à une itération i, contient des noeuds $U(a_i)$ dont les branches sont les contraintes ou les variables de cet ensemble. Un IIS obtenu par l'algorithme d'insertion correspond alors à un parcours sans retour-arrière de l'arbre où chaque branche visitée est une contrainte ou variable de cet IIS. L'algorithme de *retour-arrière* est une variante de l'algorithme d'insertion qui explore cet arbre en faisant un retour-arrière lorsqu'un IIS est obtenu, où une borne rencontrée. Ainsi, cet algorithme permet de trouver le plus court parcours de l'arbre menant à un IIS. Cet IIS correspond donc au plus petit IIS pouvant être obtenu par l'algorithme d'insertion. Finalement, on peut utiliser la taille du dernier IIS trouvé comme borne supérieure à la profondeur d'exploration de l'arbre. Ainsi, on évite d'explorer les branches menant à un parcours de longueur égale ou supérieure au meilleur trouvé.

Soit d une profondeur dans l'arbre de recherche, correspondant au nombre de contraintes ou variables de poids $|\mathcal{E}|$. On note $B(d)$ le dernier noeud visité à cette profondeur. Ce noeud contient un ensemble de branches associées aux contraintes ou aux variables de l'ensemble $U(a)$ obtenu à cette itération. On note, par ailleurs, $N(d) \subseteq B(d)$ l'ensemble des branches de $B(d)$ qui n'ont pas encore été explorées. Au départ, l'algorithme de retour-arrière initialise le poids des contraintes ou variables de $\mathcal{E}$ à 1, la profondeur d à 0, et K à $\mathcal{E}$. Alors que K n'est pas nécessairement un IIS, $|K|$ est utilisé comme borne supérieure pour d. Ainsi, à chaque itération i, si $d < |K|$ une affectation a_i est obtenue avec *procMWIC*. Si $f(w, a_i) < |\mathcal{E}|$, d est incrémentée de 1 et l'ensemble $U(a_i)$ est copié dans $N(d)$ et $B(d)$. Sinon, l'algorithme a trouvé un plus petit IIS K. Par contre, si $d = |K|$, le parcours courant dans l'arbre de recherche peut, dans le meilleur cas, donner un IIS de la même taille que K. L'algorithme doit donc faire un retour-arrière et choisir une nouvelle branche à explorer. Ainsi, le poids des contraintes ou variables de $B(d)$ est ré-initialisé à 1, pour effacer les changements faits à la profondeur d, et d est

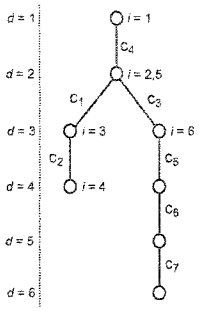

Figure 3.3 Illustration de l'algorithme de retour-arrière

décrémentée de 1. Ce processus est répété, jusqu'à ce que l'algorithme obtienne une nouvelle profondeur d contenant une branche inexplorée (i.e. $N(d) \neq \emptyset$), ou qu'aucune telle profondeur soit trouvée (i.e $d = 0$). Si l'algorithme a identifié une profondeur d pour laquelle une branche n'a pas été explorée, le poids de la contrainte ou variable, correspondant à cette branche, est fixé à $|\mathcal{E}|$, et celui des autres variables ou contraintes de $B(d)$ est fixé à 0. Finalement, l'algorithme se termine lorsque toutes les branches du premier niveau ont été explorées (i.e. $d = 0$).

La figure 3.3 illustre la détection d'un IIS de contraintes pour le problème de 2-coloriage du graphe de la figure 3.1. Cette figure montre l'arbre de recherche d'une détection possible. À la gauche de l'arbre se trouve la profondeur des noeuds de l'arbre. De plus, à côté de chaque noeud est inscrit les itérations pour lesquelles ce noeud a été modifié. Finalement, à côté de chaque branche se trouve la contrainte qui a été conservée lors de l'exploration de cette branche. Ainsi, à la première itération, l'affectation a_1 obtenue par *procMWIC* donne $U(a_1) = \{c_4\}$, et le poids de c_4 est fixé à $\mathcal{C} = 7$. L'ensemble $U(a_2)$ doit alors contenir une contrainte de $\{c_1, c_2\}$ et une autre de $\{c_3, c_5, c_6, c_7\}$, par exemple $U(a_2) = \{c_1, c_3\}$. Si la branche correspondant à c_1 est d'abord explorée, le poids de c_1 est fixé à 7 et celui de c_3 à 0. À l'itération 3, $U(a_3) = \{c_2\}$ et le poids de c_2 est fixé à 7. Ensuite, *procMWIC* retourne une affectation a_4 de poids $f(w, a_4) = |\mathcal{E}|$ et l'IIS minimum

Algorithme 9 Algorithme de retour-arrière

Entrée: Un CN incohérent P;
Sortie : Un IIS de contraintes ou de variables K.

Initialisation
pour $e \in \mathcal{E}$ **faire**
  $w(e) \leftarrow 1$;
fin pour
$d \leftarrow 0$;
$K \leftarrow \mathcal{E}$;

Construction
répéter
  si $d < |K|$ **alors**
    $a \leftarrow$ procMWIC(P, w);
    si $f(w, a) < |\mathcal{E}|$ **alors**
      $d \leftarrow d + 1$;
      $B(d) \leftarrow N(d) \leftarrow \{e \in U(a) \mid w(e) = 1\}$;
    sinon
      $K \leftarrow \{e \mid w(e) = |\mathcal{E}|\}$;
    fin si
  sinon
    répéter
      pour tout $e \in B(d)$ **faire**
        $w(e) \leftarrow 1$;
      fin pour
      $d \leftarrow d - 1$;
    juqu'à ce que $N(d) \neq \emptyset$ ou $d = 0$
  fin si
  si $0 < d < |K|$ **alors**
    Choisir un élément $e \in N(d)$;
    $N(d) \leftarrow N(d) \setminus \{e\}$;
    $w(e) \leftarrow |\mathcal{E}|$;
    pour tout $e' \in B(d)$, $e' \neq e$ **faire**
      $w(e') \leftarrow 0$;
    fin pour
  fin si
juqu'à ce que $d = 0$

$K = \{c_1, c_2, c_4\}$ est détecté. La profondeur maximale de l'arbre à laquelle un IIS peut être trouvé devient alors $|K| = 3$. Puisque $d > |K|$, l'algorithme retourne à la dernière branche inexplorée, se trouvant à $d = 2$, et ré-initialise le poids de c_2 à 1. L'algorithme explore alors la branche correspondant à c_3, en fixant son poids à 7 et celui de c_1 à 0. L'affectation a_5 donne ensuite l'ensemble $U(a_5) = \{c_5\}$. Par ailleurs, puisqu'aucun IIS n'est trouvé à l'itération 6, l'algorithme retourne en arrière à la recherche d'une branche inexplorée. Cependant, comme toutes les branches ont été explorées, d tombe à 0 et l'algorithme retourne l'IIS minimum $K = \{c_1, c_2, c_4\}$.

Propriété 3.3.2 *Soit P un CN incohérent et $\mathcal{I}$ l'ensemble des IIS de P pouvant être obtenus avec l'algorithme d'insertion. L'algorithme de retour-arrière produit, en un nombre fini d'itérations, un IIS $K \in \mathcal{I}$ de cardinalité minimum.*

Propriété 3.3.3 *Soit P un CN incohérent et $\mathcal{I}$ l'ensemble des IIS de P pouvant être obtenus avec l'algorithme d'insertion. Le nombre maximum d'itérations i_{max} requis par l'algorithme de retour-arrière pour obtenir un IIS minimum $K \in \mathcal{I}$ est borné par:*

$$i_{max} \leq \sum_{i=1}^{|\mathcal{I}|} |K_i| + 1$$

Preuve On constate que, pour chaque noeud $N(d)$ à une profondeur d, l'algorithme ne conserve un IIS $K_i \in \mathcal{I}$ que si $N(d)$ contient une seule contrainte ou variable $e \in K_i$. Ainsi, K_i est conservé si la branche de e est explorée, sinon K_i est détruit. Il existe donc un seul parcours menant à chaque K_i, et le nombre total de parcours de l'arbre est $|\mathcal{I}|$. Par ailleurs, la longueur du parcours menant à K_i est au plus $|K_i| + 1$, puisque qu'une contrainte ou variable de K_i est conservée à chaque itération, et qu'il faut une itération supplémentaire pour détecter l'IIS. Finalement,

le nombre maximum d'itérations pour obtenir n'importe quel IIS $K \in \mathcal{I}$ est donc inférieur ou égal à la somme de la longueur des $|\mathcal{I}|$ parcours de l'arbre.

On remarque que, comme l'algorithme d'insertion, l'algorithme de retour-arrière ne permet pas d'obtenir tous les IIS de P. Considérons à nouveau la détection d'un IIS de contraintes pour le problème de 2-coloriage de la figure 3.2. La première affectation viole nécessairement les trois contraintes du centre[4], formant l'unique IIS minimum de P. Le premier noeud de l'arbre contient donc trois branches, correspondant à ces contraintes. Ainsi, peu importe la branche explorée à ce niveau, l'IIS minimum sera détruit. Par ailleurs, on constate que l'algorithme de retour-arrière obtient un IIS minimum en un nombre exponentiel d'itérations, seulement si P contient un nombre exponentiel d'IIS. Or, il arrive souvent qu'un CN incohérent de grande taille ne contienne qu'un nombre réduit d'IIS. Dans ce cas, l'algorithme de retour-arrière trouve un IIS minimum très rapidement. D'autre part, plus un noeud $B(d)$ contient de branches, plus il y aura d'IIS détruits lors de l'exploration de ce noeud. Ainsi, plus d augmente, moins les noeuds $B(d)$ auront de branches. Ce principe permet de limiter la taille d'un arbre dont les premiers noeuds contiennent un grand nombre de branches.

3.3.3 Algorithme de pré-filtrage

Lors de la détection d'IIS dans un CN incohérent de grande taille, il est souvent utile de rapidement éliminer le plus possible de contraintes et de variables, afin d'en laisser moins pour l'algorithme de détection. L'algorithme de *pré-filtrage*, présenté dans cette section, est une variante de l'algorithme d'insertion, utilisée comme pré-traitement sur le CN avant de lui appliquer un des algorithmes de détection

[4]En gras dans la figure.

précédents.

À chaque itération i, *procMWIC* retourne une affectation a_i. À la différence de l'algorithme d'insertion, le poids de toutes les contraintes ou variables de $U(a_i)$ est ensuite fixé à $|\mathcal{E}|$. On peut alors obtenir rapidement un ensemble incohérent K, sur lequel est ensuite appliqué l'algorithme de détection. Par ailleurs, comme le poids d'au moins une contrainte ou variable de chaque IIS est fixé $|\mathcal{E}|$, à chaque itération, un petit IIS a plus de chance d'être isolé, par cet algorithme, qu'un plus gros. Ainsi, l'algorithme de pré-filtrage agit également comme une heuristique pour isoler de plus petits IIS.

Algorithme 10 Algorithme de pré-filtrage

Entrée: Un CN incohérent P;
Sortie : Un ensemble incohérent de contraintes ou de variables K.

Initialisation
pour tout $e \in \mathcal{E}$ **faire**
  $w(e) \leftarrow 1$;
fin pour

Construction
répéter
  $a \leftarrow$ procMWIC(P, w);
  si $f(w, a) < |\mathcal{E}|$ **alors**
    pour tout $e \in U(a)$ **faire**
      $w(e) \leftarrow |\mathcal{E}|$;
    fin pour
  fin si
juqu'à ce que $f(w, a) \geq |\mathcal{E}|$

Extraction
$K \leftarrow \{e \mid w(e) = |\mathcal{E}|\}$;

Illustrons, une fois de plus, l'algorithme de pré-filtrage sur la détection d'un IIS de contraintes pour le problème de 2-coloriage du graphe de la figure 3.1. Après avoir initialisé le poids des contraintes de $\mathcal{E}$ à 1, une affectation a_1 est obtenue avec *procMWIC*, donnant $U(a_1) = \{c_4\}$. Le poids de c_4 est alors fixé à $\mathcal{C} =$

7, et l'ensemble $U(a_2)$ doit contenir une contrainte de $\{c_1, c_2\}$ et une autre de $\{c_3, c_5, c_6, c_7\}$, par exemple $U(a_2) = \{c_1, c_3\}$. Le poids de c_1 et c_3 est ensuite fixé à 7, et $U(a_3)$ doit alors contenir c_2 et une autre contrainte de $\{c_5, c_6, c_7\}$, par exemple c_5. Le poids de ces deux contraintes est alors fixé à 7, et comme l'affectation suivante a_4 a un poids $f(w, a_4) = |\mathcal{E}|$, l'algorithme retourne $K = \{c_1, c_2, c_3, c_4, c_5\}$. Alors que cet ensemble incohérent n'est pas un IIS, on remarque qu'il ne contient que l'IIS minimum $\{c_1, c_2, c_4\}$.

Soit K un IIS minimum de P, il est facile de constater que l'algorithme de pré-filtrage se termine, dans le pire cas, après $|K| + 1$ itérations. Par ailleurs, cet algorithme isole cet IIS minimum si la détection de tout autre IIS de P nécessite plus de $|K|$ itérations. Ainsi, dans l'exemple précédent, l'IIS minimum a été isolé en quatre itérations, alors qu'il aurait fallu cinq itérations pour isoler l'autre IIS. Il existe, cependant, des cas où l'algorithme de pré-filtrage ne parvient pas à isoler l'IIS minimum. La détection d'un IIS de contraintes pour le problème de 2-coloriage du graphe de la figure 3.4 est un tel cas. La première affectation viole les cinq contraintes au centre du graphe[5], formant un IIS qui n'est pas minimum. Le poids de ces contraintes est alors fixé à $\mathcal{C} = 25$, et cet ensemble incohérent de contraintes est ensuite détecté à l'itération suivante. L'algorithme retourne ainsi un ensemble K qui ne contenant pas un IIS minimum.

Pour tous les algorithmes de détection présentés dans ce chapitre, il est plus facile d'obtenir un IIS de P, si P ne contient que cet IIS. Dans certains cas, l'algorithme de pré-filtrage retourne cependant un ensemble K contenant plusieurs IIS. Considérons, par exemple, la détection d'un IIS de variables dans le problème de 2-coloriage de la figure 3.5. À chaque itération i, $U(a_i)$ doit contenir une variable de chacun des trois IIS minimum du problème, correspondant aux triangles de

[5]En gras dans la figure.

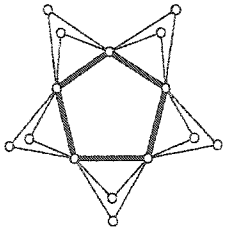

Figure 3.4 Une instance où l'algorithme de pré-filtrage n'isole pas d'IIS minimum de contraintes pour $k = 2$

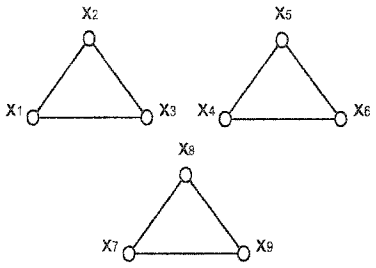

Figure 3.5 Une instance où l'algorithme de pré-filtrage isole plus d'un IIS pour $k = 2$

la figure. Ainsi, à l'itération 1, on peut avoir $U(a_1) = \{x_1, x_4, x_7\}$, et le poids de ces trois variables est fixé à $\mathcal{X} = 9$. Ensuite, à l'itération 2, a_2 peut donner $U(a_2) = \{x_2, x_5, x_8\}$, et le poids de ces variables devient 9. Finalement, à l'itération 3, $U(a_3)$ contient nécessairement les trois dernières variables: $U(a_3) = \{x_3, x_6, x_9\}$. Cependant, lorsque le poids de ces variables est fixé à 9, les trois IIS minimum se forment dans K.

On remarque, de l'exemple précédent, que les IIS contenus dans K sont formés lorsque le poids des contraintes ou variables du dernier ensemble $U(a_i)$ est fixé à $|\mathcal{E}|$. L'algorithme 11 modifie l'algorithme de pré-filtrage, afin d'éviter de former plusieurs IIS lors de cette dernière modification des poids. Ainsi, à une itération i, ce nouvel algorithme conserve dans T les contraintes ou variables de $U(a_i)$. Ensuite, si *procMWIC* retourne une affectation a_{i+1} de coût $f(w, a_{i+1}) \geq |\mathcal{E}|$, le poids des contraintes ou variables de T est ré-initialisé à 1. Finalement, les contraintes ou

variables de T reçoivent, une à une, un poids de $|\mathcal{E}|$, jusqu'à ce qu'une nouvelle affectation de coût supérieur ou égal à $|\mathcal{E}|$ soit obtenue.

Algorithme 11 Algorithme modifié de pré-filtrage

Entrée: Un CN incohérent P;
Sortie : Un ensemble incohérent de contraintes ou de variables K.

Initialisation
pour tout $e \in \mathcal{E}$ **faire**
  $w(e) \leftarrow 1$;
fin pour

Construction
répéter
  $a \leftarrow$ procMWIC(P, w);
  si $f(w, a) < |\mathcal{E}|$ **alors**
    $T \leftarrow U(a)$;
    pour tout $e \in T$ **faire**
      $w(e) \leftarrow |\mathcal{E}|$;
    fin pour
  fin si
juqu'à ce que $f(w, a) \geq |\mathcal{E}|$

Isolation
pour tout $e \in T$ **faire**
  $w(e) \leftarrow 1$;
fin pour
répéter
  Choisir un élément $e \in T$;
  $T \leftarrow T \setminus \{e\}$;
  $w(e) \leftarrow |\mathcal{E}|$;
  $a \leftarrow$ procMWIC(P, w);
juqu'à ce que $f(w, a) \geq |\mathcal{E}|$

Extraction
$K \leftarrow \{e \mid w(e) = |\mathcal{E}|\}$;

En revenant à l'exemple précédent, l'algorithme modifié de pré-filtrage permet d'isoler un seul IIS minimum, au lieu des trois. Ainsi, à l'itération 3, le contenu de $U(a_3) = \{x_3, x_6, x_9\}$ est conservé dans T, et lorsque *procMWIC* obtient une affectation a_4 de coût $f(w, a_4) \geq |\mathcal{E}|$, le poids des variables de T est ré-initialisé à 1.

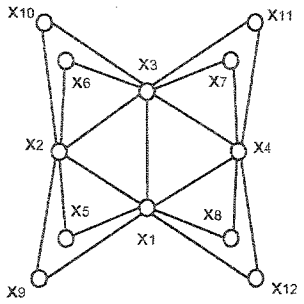

Figure 3.6 Une instance où l'algorithme de pré-filtrage modifé isole plus d'un IIS de variables pour $k = 2$

Ensuite, le poids d'une variable de T est fixé à 9, par exemple x_3. Une affectation a_5 de coût $f(w, a_5) \geq |\mathcal{E}|$ est alors obtenue, et l'algorithme retourne l'ensemble $K = \{x_1, x_2, x_3, x_4, x_5, x_7, x_8\}$, qui contient un seul IIS minimum. Il existe, par contre, des cas pour lesquels l'algorithme modifié de pré-filtrage ne permet pas d'isoler un seul IIS. Soit, par exemple, la détection d'un IIS de variables dans le problème de 2-coloriage du graphe de la figure 3.6. L'affectation a_1 donne, tout d'abord, un ensemble $U(a_1)$ valant soit $\{x_1, x_3\}$ ou $\{x_2, x_4\}$. Si, par exemple, $U(a_1) = \{x_2, x_4\}$, on aura $U(a_2) = \{x_1, x_3\}$. Le poids de x_1 et x_3 est ensuite fixé à $|\mathcal{X}| = 12$, puis ré-initialisé à 1, lorsque une affectation a_3 de coût $f(w, a_3) \geq |\mathcal{E}|$ est obtenue. Le poids d'une de ces variables, par exemple x_1, est ensuite à nouveau fixé à 12. Finalement, lorsque le poids de x_3 est également fixé à 12, deux IIS minimum sont formés.

On remarque, de plus, que la modification faite à l'algorithme de pré-filtrage augmente le nombre d'itérations, dans le pire cas, de $|\mathcal{E}| + 1$. Cependant, comme les ensembles $U(a_i)$ obtenus à chaque itération i contiennent généralement une faible proportion des contraintes ou des variables de $\mathcal{E}$, cette modification entraîne, la plupart du temps, un petit nombre d'itérations supplémentaires. Finalement, sachant que les IIS sont formés lorsqu'un dernier poids est fixé à $|\mathcal{E}|$, on peut utiliser une heuristique, comme celle présentée à la section suivante, pour d'abord modifier le

poids de la contrainte ou variable située à proximité de celles ayant un poids de $|\mathcal{E}|$. On peut ainsi obtenir de plus petits IIS.

3.3.4 Heuristique de poids du voisinage

L'importance de trouver des petits IIS a déjà été établie par plusieurs travaux antérieurs (*voir chapitre 2*). Par exemple (Greenberg, 1992) a montré que les IIS contenant le moins de contraintes de rangée étaient les plus utiles au diagnostic de programmes linéaires irréalisables. Ainsi, dans (Chinneck, 1997 (1)), on a utilisé une heuristique favorisant le retrait des contraintes de rangées d'abord, ou l'ajout de ces contraintes en dernier. Par ailleurs, dans (Herrmann et Hertz, 2002), Herrmann et Hertz ont montré que la détection de sous-graphes critiques dans un graphe G, contenant le moins possible de sommets, permettait d'aider une méthode exacte à déterminer le nombre chromatique de G. Ces derniers ont également proposé une heuristique qui retire de G, à chaque itération, le sommet ayant le plus petit degré (i.e. nombre de sommets voisins) dans G, afin d'obtenir un sous-graphe critique le plus dense possible. Il a été vu, précédemment, que l'algorithme de *hitting set* et l'algorithme de retour-arrière permettait d'obtenir des IIS minimum[6]. Par ailleurs, il a été montré que l'algorithme de pré-filtrage agit comme une heuristique pour isoler de petits IIS. Cette section propose une autre heuristique qui utilise l'information contenue dans les poids pour obtenir des IIS les plus petits possible.

La plupart des algorithmes présentés dans ce chapitre doivent, durant la détection d'un IIS, choisir une contrainte ou une variable d'un ensemble, afin de modifier son poids. Il a été montré, par ailleurs, que ce choix déterminait quel IIS est obtenu par l'algorithme. L'heuristique de *poids de voisinage* utilise le poids de voisinage

[6]L'algorithme de retour-arrière retourne un IIS minimum parmi ceux pouvant être obtenus par l'algorithme d'insertion.

des contraintes et des variables pour faire un choix permettant d'obtenir un plus petit IIS:

Définition Soit un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ et w une fonction qui associe un poids $w(c)$ à chaque contrainte $c \in \mathcal{C}$. Le poids de voisinage d'une contrainte $c \in \mathcal{C}$ est donné par:

$$W_{\mathcal{C}}(c) = \sum_{x \in \mathcal{X}(c)} \left(\sum_{\substack{c' \in \mathcal{C}(x) \\ c' \neq c}} w(c') \right)$$

Définition Soit un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ et w une fonction qui associe un poids $w(x)$ à chaque variable $x \in \mathcal{X}$. Le poids de voisinage d'une variable $x \in \mathcal{X}$ est donné par:

$$W_{\mathcal{X}}(x) = \sum_{c \in \mathcal{C}(x)} \left(\sum_{\substack{x' \in \mathcal{X}(c) \\ x' \neq x}} w(x') \right)$$

Un voisin d'une contrainte $c \in \mathcal{C}$ est donc défini comme une autre contrainte $c' \in \mathcal{C}$ agissant sur une même variable que c (i.e $\mathcal{X}(c) \cap \mathcal{X}(c') \neq \emptyset$). Ainsi, le poids du voisinage de c correspond à la somme, pour toutes les contraintes c' voisines de c, des poids $w(c')$. De même, un voisin d'une variable $x \in \mathcal{X}$ est une autre variable $x' \in \mathcal{X}$ impliquée dans la même contrainte que x (i.e. $\mathcal{C}(x) \cap \mathcal{C}(x') \neq \emptyset$). Le poids du voisinage de x est alors la somme, pour les variables x' voisines de x, des poids $w(x')$. Par ailleurs, dans le cas où les poids des contraintes ou des variables valent 1 ou 0, le poids de voisinage permet d'estimer la densité des contraintes d'une région de P. Le poids de voisinage d'une contrainte ou variable correspond alors à son nombre de voisins. Ainsi, une contrainte ou variable, située dans une région plus dense de P, aura généralement plus de voisins qu'une autre située dans une région moins dense. De plus, lorsque certains poids ont été fixés à $|\mathcal{E}|$, le poids de

voisinage d'une contrainte ou d'une variable est un indice de sa proximité d'un IIS en construction. En effet, comme l'IIS en construction est formé des contraintes ou variables ayant un poids de $|\mathcal{E}|$, la valeur du poids de voisinage est grandement augmentée par la présence d'une telle contrainte ou variable dans le voisinage.

Le poids de voisinage peut alors être utilisé comme suit. Lorsqu'on doit retirer une contrainte ou une variable de P, on choisit normalement celle ayant le plus petit poids de voisinage, afin de ne pas détruire les IIS les plus denses de P. Cette stratégie se base sur le fait que, pour l'algorithme de retrait, l'IIS obtenu est celui ayant une première contrainte ou variable retirée en dernier. L'heuristique de poids de voisinage effectue donc des retraits dans les régions les moins denses de P d'abord. Par ailleurs, lorsqu'on doit conserver une contrainte ou une variable (i.e. fixer son poids à $|\mathcal{E}|$), on doit faire l'inverse. L'heuristique de poids de voisinage conserve d'abord la contrainte ou variable ayant le plus grand poids de voisinage, afin de garder l'IIS en construction le plus dense possible.

Illustrons l'heuristique de poids de voisinage à l'aide de deux exemples. Considérons, tout d'abord, le CN associé au problème de 2-coloriage du graphe à la gauche de la figure 3.7. Ce CN montre le poids de voisinage des contraintes, après que leur poids ait été initialisé à 1. Ainsi, la contrainte au centre du CN a un poids de voisinage de 4 parce qu'elle partage une variable avec quatre autres contraintes de poids 1. Supposons que l'on veuille obtenir un IIS de contraintes à l'aide de l'algorithme de retrait, l'heuristique de poids de voisinage choisit une contrainte c, ayant la plus petite valeur pour $W_{\mathcal{C}}(c)$. La première contrainte retirée doit donc être une des deux contraintes avec $W_{\mathcal{C}}(c) = 2$. On remarque que ce retrait détruit un des deux IIS du CN, conservant l'IIS minimum. Par ailleurs, la partie droite de la figure 3.7 montre les poids de voisinage des variables du même CN, après deux itérations de l'algorithme d'insertion. Les sommets noircis de la figure correspondent aux variables dont le poids a été fixé à $|\mathcal{X}| = 6$, et les autres aux variables

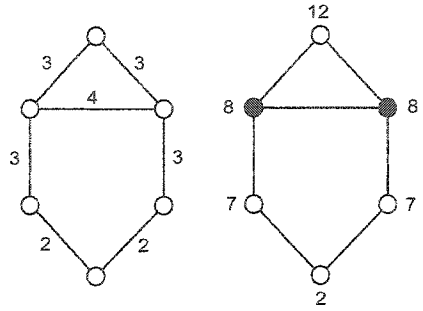

Figure 3.7 Exemples du poids de voisinage de contraintes et de variables

de poids 1. L'itération 3 de l'algorithme d'insertion doit alors donner un ensemble $U(a_3)$ contenant une variable de chacun des deux IIS du CN, par exemple la variable de poids de voisinage 12 et une autre de poids de voisinage 7. L'heuristique de poids de voisinage choisit alors de conserver la variable x ayant la plus grande valeur pour $W_{\mathcal{X}}(x)$. Le poids de la variable avec $W_{\mathcal{X}}(x) = 12$ est donc fixé à $|\mathcal{X}| = 6$ et celui de l'autre variable à 0, et l'algorithme d'insertion obtient alors l'IIS minimum à l'itération suivante.

3.3.5 Borne inférieure sur la taille d'IIS minimum

Il a été montré à la section 7 qu'une borne inférieure sur la taille d'IIS minimum d'un CN incohérent pouvait être obtenue en arrêtant l'algorithme de *hitting set* à n'importe quelle itération et utiliser la taille du dernier HS retourné par *procMHS* comme borne. Cependant, cette borne est seulement valide dans le cas où *procMHS* obtient des HS de cardinalité minimum. Cette section présente un autre algorithme permettant d'obtenir une borne inférieure sur la taille d'IIS minimum d'un CN incohérent.

L'algorithme 12 prend en paramètre un CN incohérent P ainsi qu'un entier $i_{\max} > 0$ correspondant au nombre maximum d'itérations allouées, et retourne une borne

Algorithme 12 Algorithme de borne inférieure sur la taille d'IIS minimum

Entrée: Un CN incohérent P et un entier $i_{\max} > 0$;
Sortie : Une borne inférieure b.

Initialisation
pour tout $e \in \mathcal{E}$ **faire**
  $\mu(e) \leftarrow 0$;
fin pour
$i \leftarrow 0$;

Calcul
tant que $i < i_{\max}$ **faire**
  pour tout $e \in \mathcal{E}$ **faire**
    $w(e) \leftarrow |\mathcal{E}|^{\mu(e)}$;
  fin pour
  $a \leftarrow \text{procMWIC}(P, w)$;
  pour tout $e \in U(a)$ **faire**
    $\mu(e) \leftarrow \mu(e) + 1$;
  fin pour
  $b = \max\big(b,\ g(\mu, i)\big)$;
fin tant que

inférieure b sur la taille d'un IIS minimum de P. De plus, cet algorithme utilise une fonction $\mu : \mathcal{E} \rightarrow \mathbb{N}^+$ qui associe à chaque contrainte ou variable $e \in \mathcal{E}$ le nombre d'itérations $\mu(e)$ pour lesquelles e a fait partie de l'ensemble $U(a)$. À chaque itération i, *procMWIC* retourne une affectation a_i donnant un ensemble $U(a_i)$ de contraintes ou de variables e pour lesquelles $\mu(e)$ est incrémentée. Une borne inférieure temporaire b' est ensuite obtenue d'une fonction g définie comme suit:

Définition Soit $\mathcal{E}$ un ensemble d'éléments, μ une fonction qui associe un entier $\mu(e)$ à chaque élément $e \in \mathcal{E}$, et soit $e_1 \geq \ldots \geq e_{|\mathcal{E}|}$ un ordre de ces éléments tel que $\mu(e_1) \geq \ldots \geq \mu(e_{|\mathcal{E}|})$. Étant donné un entier i, $g(\mu, i)$ est le plus petit entier l tel que:

$$\sum_{j=1}^{l} \mu(e_j) \geq i$$

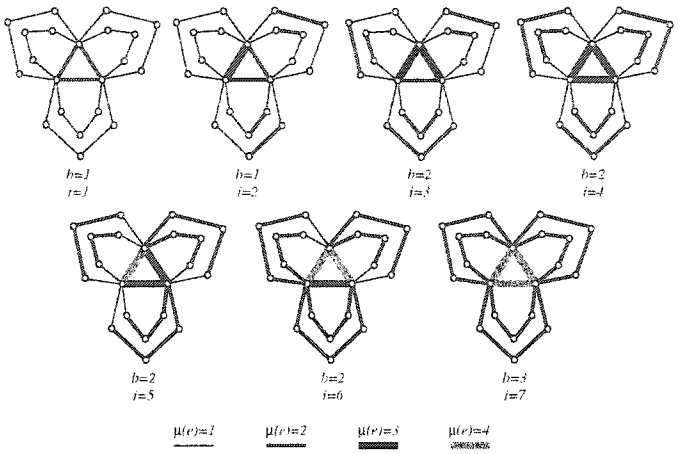

Figure 3.8 Illustration de l'algorithme 12

Enfin, comme la borne inférieure b' retournée par g peut décroître à n'importe quelle itération, la meilleure borne inférieure b est alors la plus grande valeur entre b et b'.

Afin d'illustrer l'algorithme 12, considérons à nouveau la détection d'un IIS de contraintes dans le problème de 2-coloriage du graphe de la figure 3.2 qui contient un IIS minimum que l'algorithme d'insertion ne peut obtenir. La figure 3.8 montre les détails des sept premières itérations de l'algorithme. Comme avant, la première affectation a_1 donne un ensemble $U(a_1)$ contenant les trois contraintes, formant un triangle au centre de la figure. Pour ces contraintes e, $\mu(e)$ est incrémenté de 1, tel que $w(e) = |\mathcal{E}|$. Comme seulement une de ces contraintes est nécessaire pour totaliser $i = 1$ dans g, la borne inférieure b est fixée à 1. Étant donné qu'une contrainte de chaque IIS est violée à chaque itération, l'affectation suivante devrait donner un ensemble $U(a_2)$ ayant une de ces trois mêmes contraintes, ainsi que quatre autres contraintes des IIS restants, comme le montre le second graphe de la figure 3.8. Ainsi, pour la contrainte de gauche du triangle on a $\mu(e) = 2$ et puisque cette valeur suffit à totaliser 2 dans g, b reste 1. La même chose se produit à l'itération 3, à l'exception que la contrainte de droite du triangle est violée, et

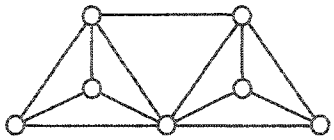

Figure 3.9 Une instance où l'algorithme 12 produit une borne inférieure sous-optimale

que deux contraintes sont nécessaires pour totaliser 3 dans g, tel que $b = 2$. À l'itération 4, les trois contraintes du triangle ont $\mu(e) = 2$, et comme seulement deux de ces contraintes sont requises pour totaliser 4, b reste 2. Les itérations 5 et 6 causent ensuite deux contraintes du triangle à avoir $\mu(e) = 3$, de telle sorte que b vaut toujours 2. Cependant, à l'itération 7, les trois contraintes du triangle ont $\mu(e) = 3$, et trois contraintes sont nécessaires pour totaliser 7 dans g (i.e. deux contraintes avec $\mu(e) = 3$ et une autre avec $\mu(e) \geq 1$). La borne inférieure b devient alors 3. Puisque b est alors égale à la taille de l'unique IIS minimum, les itérations suivantes sont inutiles. Ainsi, $i_{\max} = 7$ est une valeur optimale du nombre d'itérations pour cet exemple.

Il existe, cependant, des cas où cet algorithme est incapable d'obtenir une borne inférieure b de la taille d'un IIS minimum. Considérons, par exemple, la détection d'IIS de variables dans le problème de 2-coloriage du graphe de la figure 3.9, dont le nombre chromatique est 4. On peut vérifier que l'algorithme retourne une borne $b = 2$, alors que n'importe quel triangle de ce graphe correspond à un IIS de trois variables.

CHAPITRE 4

ALGORITHMES HEURISTIQUES DE DÉTECTION D'IIS

Les algorithmes présentés au chapitre précédent ne sont pas très utiles en pratique à cause de leur complexité. Ainsi, tous ces algorithmes utilisent la procédure *procMWIC* pour résoudre de manière exacte le MWCSP, dans le cas de la détection d'IIS de contraintes, ou le MPWCSP, dans le cas de la détection d'IIS de variables. Il a cependant été démontré que le MWCSP et MPWCSP sont deux problèmes NP-difficiles. Par ailleurs, l'algorithme de *hitting set* utilise, en plus, la procédure *procMHS* qui résoud de manière exacte le MHS, un autre problème NP-difficile. Les algorithmes de détection sont donc plus complexes que ces procédures appelées à chaque itération. La situation est encore pire dans le cas des algorithmes de *hitting set* et de retour-arrière qui prennent un nombre exponentiel d'itérations pour trouver un IIS. La section suivante porte sur l'utilisation de méta-heuristiques pour résoudre les problèmes NP-difficiles sous-jacents aux algorithmes de détection, afin de réduire leur complexité.

4.1 Méta-heuristiques pour les problèmes de satisfaction

Le MWCSP et le MPWCSP, présentés au chapitre précédent, sont deux exemples de problèmes d'optimisation combinatoire (Papadimitriou et Steiglitz, 1982):

Définition Une instance d'un problème d'optimisation combinatoire (OC) est une paire $(\mathcal{S}, c)$ où $\mathcal{S}$ est une ensemble de configurations réalisables de cardinalité finie, appelé l'espace de recherche, et $f : \mathcal{S} \rightarrow \mathbb{R}$ une fonction de coût qui associe une

valeur réelle à chaque configuration $s \in \mathcal{S}$. Le but est de trouver une configuration $s^* \in \mathcal{S}$ telle que:

$$f(s^*) \leq f(s) \quad \forall\, s \in \mathcal{S}$$

Une telle configuration est appelé une solution *globalement optimale* de l'instance.

Ainsi, pour le MWCSP, une configuration $s \in S$ correspond à une affectation complète a, et f à la somme $f_C(w, a)$ des poids des contraintes violées par a. Par ailleurs, dans le cas du MPWCSP, une configuration s correspond à une affectation légale a, et f à la somme $f_{\mathcal{X}}(w, a)$ des poids des variables n'étant pas instanciées par a. Comme il a été montré, la complexité de calculer une solution globalement optimale n'est pas acceptable pour plusieurs problèmes d'optimisation combinatoire, comme le MWCSP et le MPWCSP. Il faut alors opter pour une méthode de recherche heuristique qui n'offre aucune garantie sur l'optimalité de la solution obtenue. Parmi les approches heuristiques, les méthodes de recherche locale (ou recherche de voisinage) sont particulièrement efficaces pour résoudre une grande variété d'OCs.

4.1.1 La recherche locale

Le principe des méthodes de recherche locale est d'optimiser localement une configuration en considérant uniquement le voisinage de cette configuration.

Définition Étant donné une instance $(\mathcal{S}, f)$ d'un OC, un voisinage est une fonction $N : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ qui associe à chaque configuration $s \in \mathcal{S}$ un sous-ensemble de configurations $N(s) \subset \mathcal{S}$ près de s selon un critère donné. Le voisinage d'une configuration s peut également être considéré comme l'ensemble de configurations pouvant être

obtenues en appliquant à s un ensemble $\mathcal{M}$ de mouvements élémentaires:

$$N(s) = \{s' \in \mathcal{S} \mid s' = \mu(s),\ \mu \in \mathcal{M}\}$$

Définition Étant donné une instance $(\mathcal{S}, f)$ d'un OC et un voisinage N, une configuration $s \in \mathcal{S}$ est un optimum local, selon N, si:

$$f(s) \leq f(s') \quad \forall\ s' \in N(s)$$

L'algorithme 13 illustre, de manière simplifiée, le principe de la recherche locale. En partant d'une configuration initiale quelconque, si une configuration s' du voisinage de la configuration courante s possède un coût inférieur à celui de s (i.e. $f(s') < f(s)$), s' devient la configuration courante. Sinon, la configuration courante est un minimum local.

Algorithme 13 Algorithme simplifié de recherche locale

Entrée: Un OC $(\mathcal{S}, f)$ et un voisinage N;
Sortie : Un minimum local s.

```
s ← une configuration quelconque dans S;
tant que ∃ s' ∈ N(s) tel que f(s') < f(s) faire
  s ← s';
fin tant que
```

Les méthodes de réparation constituent une approche de recherche locale possible pour résoudre le CSP. Ces méthodes réparent une affectation initialement incohérente en diminuant itérativement le nombre de conflits. L'algorithme de *Moindre-conflit* (MCH) est une méthode de réparation qui résoud le MCSP. Son fonctionnement est simple. Une affectation initiale a est d'abord choisie, par exemple de manière aléatoire. L'algorithme choisit ensuite une variable impliquée dans une contrainte violée par a, et lui assigne une nouvelle valeur de son domaine, telle

que l'affectation résultante a' viole le moins possible de contraintes[1]. Si a' viole un nombre inférieur ou égal de contraintes à a, a' devient l'affectation courante de la prochaine itération. Sinon, l'algorithme conserve a, et modifie une nouvelle variable à l'itération suivante. L'algorithme se termine lorsqu'une affectation légale est trouvée, ou après avoir fait un certain nombre d'itérations.

4.1.1.1 Algorithme pour le MWCSP

Dans le cas où les contraintes du CN ont une pondération, il ne suffit plus de minimiser le nombre de contraintes violées. La fonction de coût à minimiser est alors la somme $f_C(w, a)$ des poids des contraintes violées par a. L'algorithme 14 est une variante du MCH pour les CN pondérés. Cet algorithme prend en paramètres un CN incohérent P, une fonction de pondération des contraintes de P, et un entier $i_{\max}$, correspondant au nombre maximum de mouvements autorisés pour la recherche locale, et retourne une affectation a^*. Cet algorithme initialise d'abord l'itération courante i à 1. De plus, une affectation complète initiale a est générée en assignant à chaque variable une valeur de son domaine choisie avec procédure *random(S)*, qui retourne un élément $s \in S$ choisi de manière équiprobable. Ensuite, pour chaque variable x impliquée dans une contrainte violée par a et chaque nouvelle valeur v du domaine de cette variable, une affectation voisine a' est obtenue en assignant v à x. Si le coût de a' est inférieur à celui de tous les voisins évalués, l'ensemble des meilleures assignations B est modifié pour ne contenir que (x, v). Si, par ailleurs, a' est de coût égal, (x, v) est ajouté à B. Après avoir ainsi évalué tous les voisins de a, l'affectation de la prochaine itération est obtenue d'une assignation (x', v') choisie aléatoirement de l'ensemble B, et l'itération courante i est incrémentée de 1. Finalement, l'algorithme se termine lorsque i dépasse $i_{\max}$.

[1]En cas d'égalité, a' est choisie au hasard parmi les meilleures trouvées.

Algorithme 14 Algorithme de recherche locale pour le MWCSP

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Une fonction de pondération w;
Un entier $i_{\max} > 0$;
Sortie : Une affectation complète a^*.

Initialisation
pour tout $x \in \mathcal{X}$ **faire**
  $a(x) \leftarrow \text{random}(D_x)$;
fin pour
$a^* \leftarrow a$;
$i \leftarrow 1$;

Recherche locale
tant que $i \leq i_{\max}$ **faire**
  $T \leftarrow \emptyset$;
  pour tout $c \in U_{\mathcal{C}}(a)$ **faire**
    $T \leftarrow T \cup \mathcal{X}(c)$;
  fin pour
  $B \leftarrow \emptyset$;
  $l \leftarrow 0$;
  pour tout $x \in T$ **faire**
    $a' \leftarrow a$;
    pour tout $v \in D_x$, $v \neq a(x)$ **faire**
      $a'(x) \leftarrow v$;
      si $f_{\mathcal{C}}(w, a') < l$ ou $B = \emptyset$ **alors**
        $B \leftarrow \{(x, v)\}$;
        $l \leftarrow f_{\mathcal{C}}(w, a')$;
      sinon si $f_{\mathcal{C}}(w, a') = l$ **alors**
        $B \leftarrow B \cup \{(x, v)\}$;
      fin si
    fin pour
  fin pour
  $(x', v') \leftarrow \text{random}(B)$;
  $a(x') \leftarrow v'$;
  si $f_{\mathcal{C}}(w, a) < f_{\mathcal{C}}(w, a^*)$ **alors**
    $a^* \leftarrow a$;
  fin si
  $i \leftarrow i + 1$;
fin tant que

On remarque quelques différences entre cet algorithme et le MCH. Tout d'abord, cet algorithme choisit la meilleure affectation voisine, même si celle-ci viole plus de contraintes. Ensuite, alors que le MCH évalue l'affectation d'une valeur à une seule variable des contraintes violées, cet algorithme considère, à chaque itération, chacune des variables impliquées dans une contrainte violée. Ainsi, le voisinage considéré par cet algorithme est plus grand que celui du MCH. Il faut donc plus de temps pour calculer le meilleur mouvement de chaque itération. Cependant, ce voisinage permet de trouver de meilleures affectations voisines, et ainsi, de converger plus rapidement vers un minimum local.

4.1.1.2 Algorithme pour le MPWCSP

Contrairement au MWCSP, une solution au MPWCSP est une affectation légale qui peut être partielle. Ainsi, en plus d'avoir une valeur de leur domaine, les variables d'une affectation peuvent également n'avoir aucune valeur. Le voisinage d'une affectation légale contient alors les affectations légales pour lesquelles une variable a une valeur différente, ou ayant une variable instanciée de plus ou de moins. On remarque, cependant, que modifier la valeur d'une variable, ou instancier une variable d'une affectation ne produit pas nécessairement une affectation légale. Considérons, par exemple, le problème de 2-coloriage du graphe de la figure 3.1. Supposons que les variables ont toutes un poids de 1, et que l'affectation initiale ne possède aucune variable instanciée: $a_1 = (-,-,-,-,-,-)$[2]. Cette affectation, nécessairement cohérente, a un coût de 6 qui correspond au nombre variables n'étant pas instanciées. L'ensemble des affectations voisines contient alors les affectations légales qui ont une seule variable instanciée. Comme les variables ont toutes le même poids, on peut choisir n'importe quelle assignation, par exemple

[2]Le symbole "-" signifie que la variable correspondante n'est pas instanciée.

$(x_1, 1)$, qui donne l'affectation $a_2 = (1, -, -, -, -, -)$ de coût 5. Suivant le même principe, la recherche locale peut, par exemple, visiter les affectations suivantes: $a_3 = (1, -, 2, -, -, -)$, $a_4 = (1, -, 2, -, 1, -)$ et $a_5 = (1, -, 2, -, 1, 2)$. Il n'est cependant plus possible d'instancier une autre variable à a_5, puisque cette affectation deviendrait alors incohérente. Par ailleurs, a_5, qui a un coût de 2, n'est pas optimale. Ainsi, l'affectation $(1, 2, 1, 2, 1, -)$ a un coût inférieur de 1. On peut toutefois choisir une assignation qui n'améliore pas la fonction de coût, par exemple $(x_3, 1)$ qui donne $a_6 = (1, -, 1, -, 1, 2)$, et ensuite obtenir une affectation optimale en instanciant x_4 avec la valeur 2.

Le voisinage qui vient d'être décrit considère l'assignation d'une nouvelle valeur à une seule variable qui peut être instanciée ou non. Ce voisinage ne comporte donc que des affectations de coût inférieur ou égal à celui de l'affectation courante. Cependant, le fait d'autoriser l'assignation d'une valeur à une variable déjà instanciée pose deux problèmes importants. D'une part, ce voisinage est beaucoup plus grand que celui tenant compte uniquement de l'assignation d'une valeur aux variables non-instanciées, et conséquemment, évaluer le meilleur voisin à chaque itération prend alors beaucoup plus temps. Ensuite, ce voisinage contient un grand nombre d'affectations de même coût que l'affectation courante, puisqu'aucune variable supplémentaire n'est instanciée. En conséquence, la recherche locale mettra plus de temps à trouver une région intéressante de l'espace de recherche défini par ce voisinage, et encore pire, aura plus de chances de cycler. En revanche, le voisinage autorisant uniquement l'instanciation d'une variable définit un espace de recherche ayant de plus importantes variations de la fonction de coût, permettant de mieux guider la recherche locale vers une région intéressante. On remarque, finalement, que ce voisinage est similaire à celui utilisé dans le cas du MWCSP, qui ne considère que l'affectation d'une nouvelle valeur à une variable d'une contrainte en conflit.

Revenant à l'exemple précédent, supposons que les affectations visitées aux quatre premières itérations sont: $a_1 = (-,-,-,-,-,-)$, $a_2 = (1,-,-,-,-,-)$, $a_3 = (1,-,2,-,-,-)$, $a_4 = (1,-,2,-,1,-)$ et $a_5 = (1,-,2,-,1,2)$. Ces affectations ont été produites en instanciant, à chaque itération, une variable supplémentaire. On remarque, par ailleurs, qu'instancier une variable supplémentaire de a_5 rend l'affectation suivante incohérente. On doit, dans ce cas, rendre l'affectation à nouveau cohérente en désinstanciant une ou plusieurs autres variables. Ainsi, une nouvelle affectation peut être générée en deux étapes:

1. **Instanciation**: on assigne une valeur à une variable non-instanciée d'une affectation légale;
2. **Réparation**: si l'affectation devient incohérente, on la rend à nouveau légale en désinstanciant une ou plusieurs autres variables impliquées dans les contraintes en conflit.

Le coût de ce mouvement est égal à la différence entre la somme des poids des variables désinstanciées, si il y en a, et le poids de la variable nouvellement instanciée. Par ailleurs, on remarque que, pour réparer l'affectation, il faut désinstancier au moins une variable (différente de la variable nouvellement instanciée) de chaque contrainte violée. Dans le cas du k-coloriage de graphe, une contrainte du CN correspondant est satisfaite si aucune ou une seule de ses variables est instanciée. Par contre, si on assigne, à une variable non-instanciée, la valeur de l'autre variable, il faut désinstancier cette seconde variable. Dans les termes du coloriage de graphe, si on colore un sommet avec une certaine couleur, il faut décolorer tous les sommets adjacents (i.e. reliés par une arête) qui ont cette même couleur. Dans l'affectation $a_4 = (1,-,2,-,1,2)$ de l'exemple précédent, les variables x_2 et x_4 ne sont pas instanciées. Les assignations du voisinage de a_4 sont donc: $(x_2,1)$, $(x_2,2)$, $(x_4,1)$ et $(x_4,2)$. Ces assignations n'ont pas toutes le même coût. Ainsi, $(x_2,1)$ a un

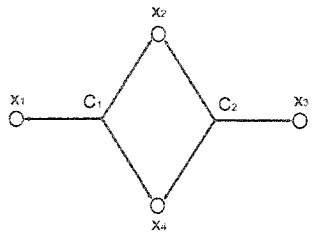

Figure 4.1 Hyper-graphe d'une instance de 3-SAT

coût nul puisqu'elle rend l'affectation incohérente et force x_1 a être désinstanciée. Le nombre de variables instanciées reste donc le même. Par contre, $(x_2, 2)$ a un coût de $+1$ car il impose aux variables x_3 et x_6 d'être désinstanciées. Finalement, $(x_4, 1)$ et $(x_4, 2)$ ont toutes les deux un coût nul puisque, dans les deux cas, une seule variable doit être désinstanciée. La recherche locale choisira donc une des trois assignations de coût nul.

On constate, par ailleurs, que le problème de choisir les variables à désinstancier, lorsque l'affectation devient incohérente, n'est pas trivial. Dans le cas où les contraintes sont binaires, par exemple le k-coloriage de graphe, il suffit de désinstancier toutes les autres variables des contraintes violées. Cependant, lorsque les contraintes ne sont pas binaires, le problème se complique. Considérons ainsi l'instance de 3-SAT[3] suivante:

$$I \;=\; (\overline{x_1} \vee x_2 \vee x_4) \;\wedge\; (x_2 \vee \overline{x_3} \vee x_4)$$

On remarque que les deux clauses de cette instance ont en commun les variables x_2 et x_4. La figure 4.1 montre l'hyper-graphe du CN correspondant à cette instance, où c_1 correspond à la première clause et c_2 à la deuxième.

Supposons que l'affectation courante soit $a_4 = (\mathrm{t}, \mathrm{f}, \mathrm{t}, -)$, où "f" correspond à la

[3]Un 3-SAT est un SAT dont les clauses ont toutes trois littéraux.

valeur de vérité négative et "t" à la valeur de vérité positive. Si on assigne à x_4 la valeur "t", toutes les clauses deviennent alors satisfaites. Cependant, s'il est impossible, pour une raison quelconque, d'assigner cette valeur à x_4, on doit lui assigner la valeur "f" qui viole simultanément les deux clauses du 3-SAT. Il faut alors désinstancier une variable de chaque contrainte. On peut ainsi désinstancier les variables x_1 et x_3 pour un coût total de +1. Toutefois, ce choix n'est pas optimal, puisqu'on peut également désinstancier uniquement la variable x_2, qui se trouve dans les deux contraintes, pour un coût nul. Ainsi, il est préférable de désinstancier des variables faisant partie du plus grand nombre possible de contraintes violées:

Propriété 4.1.1 *Soit un CN $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, une affectation partielle légale a, et une assignation (x, v) donnant une affectation incohérente a'. Soit, de plus, l'ensemble $U_{\mathcal{C}}(a')$ des contraintes violées par a'. Si chacune des contraintes de $\mathcal{C}$ peut être satisfaite en désinstanciant n'importe laquelle de ses variables, l'ensemble des variables à désinstancier dans a' dont la somme est minimale consiste en un* hitting set *minimum de la collection $\{\mathcal{X}(c) \setminus \{x\} \mid c \in U_{\mathcal{C}}(a')\}$.*

Cette propriété tient pour tous les CN dont les contraintes sont toujours satisfaites lorsque n'importe laquelle de leurs variables est désinstanciée (e.g. k-coloriage et SAT). On se rappelle, par ailleurs, que déterminer un *hitting set* minimum est un problème NP-difficile. L'algorithme 15 est une procédure qui prend en paramètre un CN incohérent P, une fonction de pondération des contraintes W, une affectation légale a et une assignation (x, v) et qui obtient, de manière gloutonne, une affectation légale a' dans laquelle x est instanciée avec la valeur v. Cet algorithme génère d'abord a' en assignant la valeur v à x. Si a' est légale (i.e. $U_{\mathcal{C}}(a') = \emptyset$), cette affectation est retournée. Sinon, l'algorithme désinstancie la variable $x' \neq x$, parmi celles impliquées dans une contrainte violée par a', qui a le plus petit poids $w(x')$. Si plusieurs de ces variables ont le même poids, la variable impliquée dans le plus

grand nombre de contraintes violées (i.e. $|\mathcal{C}(x') \cap U_{\mathcal{C}}(a')|$) est choisie. Finalement, s'il existe plusieurs de ces variables, contenues dans B, la variable à désinstancier est choisie aléatoirement parmi celles-ci. Ce processus est répété jusqu'à ce que a' soit une affectation légale.

Dans l'exemple du 3-SAT précédent, l'application de (x_4, f) à $a_4 = (\mathrm{t}, \mathrm{f}, \mathrm{t}, -)$ viole les deux clauses. Ainsi, l'ensemble des variables pouvant être désinstanciées est $\{x_1, x_2, x_3\}$. Supposons que ces variables ont toutes un poids de 1, l'algorithme désinstancie d'abord la variable impliquée dans le plus de contraintes violées, soit x_2. Comme l'affectation résultante est légale, l'algorithme se termine alors. On remarque que, dans ce cas, l'algorithme a effectivement désinstancié l'ensemble de poids minimum de variables rendant l'affectation à nouveau légale.

On possède maintenant tous les éléments nécessaires pour implanter une méta-heuristique de recherche locale pour le MPWCSP. L'algorithme 16 est une telle implantation, similaire à celle proposée pour le MWCSP. Cet algorithme prend en paramètre un CN incohérent P, un fonction de pondération des variables w, et un entier $i_{\max}$, correspondant au nombre maximum de mouvements autorisés pour la recherche locale, et retourne une affectation légale a^*. Pour commencer, l'algorithme génère une affectation a dans laquelle aucune variable n'est instanciée, et initialise l'itération courante i à 1. Ensuite, pour chaque variable x non-instanciée (i.e. $x \in U_{\mathcal{X}}(a)$) et chaque valeur v du domaine de cette variable, une procédure *procAssigne* retourne une affectation légale a' dans laquelle x est instanciée avec la valeur v. L'algorithme 15, qui vient d'être présenté, est une implantation de cette procédure qui obtient de manière gloutonne une affectation a' dont le coût $f_{\mathcal{X}}(w, a')$ est le plus faible possible. Si le coût de a' est inférieur à celui de tous les voisins évalués, l'ensemble des meilleures assignations B est modifié pour ne contenir que (x, v). Si, a' est de coût égal, (x, v) est ajouté à B. Après avoir ainsi évalué tous les voisins de a, l'affectation de la prochaine itération est obtenue de *procAssigne* avec

Algorithme 15 Algorithme glouton d'assignation pour le MPWCSP

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Une fonction de pondération w;
Une affectation légale a;
Une assignation (x, v);
Sortie : Une affectation légale a'.

Assignation
$a' \leftarrow a$;
$a'(x) \leftarrow v$;

Réparation
tant que $U_{\mathcal{C}}(a') \neq \emptyset$ **faire**
  $T \leftarrow \emptyset$;
  pour tout $c \in U_{\mathcal{C}}(a')$ **faire**
    $T \leftarrow T \cup \mathcal{X}(c)$;
  fin pour
  $B \leftarrow \emptyset$;
  $l \leftarrow 0$;
  $m \leftarrow 0$;
  pour tout $x' \in T$, $x' \neq x$ **faire**
    $t \leftarrow |\mathcal{C}(x') \cap U_{\mathcal{C}}(a')|$;
    si $w(x') < l$ ou $\mathcal{B} = \emptyset$ **alors**
      $B \leftarrow \{x'\}$;
      $l \leftarrow w(x')$;
      $m \leftarrow t$;
    sinon si $w(x') = l$ **alors**
      si $t > m$ **alors**
        $B \leftarrow \{x'\}$;
        $m \leftarrow t$;
      sinon si $t = m$ **alors**
        $B \leftarrow B \cup \{x'\}$;
      fin si
    fin si
  fin pour
  $x' \leftarrow$ random(B);
  $a'(x') \leftarrow$ aucune valeur;
fin tant que

un assignation (x', v'), choisie aléatoirement de B. Finalement, l'itération courante i est incrémentée de 1, et l'algorithme se termine lorsque i dépasse $i_{\max}$.

Algorithme 16 Algorithme de recherche locale pour le MPWCSP

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Une fonction de pondération w;
Un entier $i_{\max} > 0$;
Sortie : Une affectation légale a^*.

Initialisation
$a^* \leftarrow a$ = affectation vide;
$i \leftarrow 1$;

Recherche locale
tant que $i \leq i_{\max}$ **faire**
  $B \leftarrow \emptyset$;
  $l \leftarrow 0$;
  pour tout $x \in U_{\mathcal{X}}(a)$ **faire**
    pour tout $v \in D_x$, $v \neq a(x)$ **faire**
      $a' \leftarrow$ procAssigne$(P, w, a, (x, v))$;
      si $f_{\mathcal{X}}(w, a') < l$ ou $B = \emptyset$ **alors**
        $B \leftarrow \{(x, v)\}$;
        $l \leftarrow f_{\mathcal{X}}(w, a')$;
      sinon si $f_{\mathcal{X}}(w, a') = l$ **alors**
        $B \leftarrow B \cup \{(x, v)\}$;
      fin si
    fin pour
  fin pour
  $(x', v') \leftarrow$ random(B);
  $a \leftarrow$ procAssigne$(P, w, a, (x', v'))$;
  si $f_{\mathcal{X}}(w, a) < f_{\mathcal{X}}(w, a^*)$ **alors**
    $a^* \leftarrow a$;
  fin si
  $i \leftarrow i + 1$;
fin tant que

4.1.2 La recherche Tabou

Les algorithmes de recherche locale qui viennent d'être présentés sont basés sur le MCH. Tout comme cette stratégie, ils ont tendance à rester bloqués dans une

région entourant un optimum local (i.e. une configuration pour laquelle aucun voisin n'améliore la fonction de coût). Après avoir visité un de ces voisins, la recherche locale risque alors de revenir à cet optimum local de coût supérieur ou égal. Afin d'éviter les cycles, il est alors nécessaire d'introduire un mécanisme empêchant de revenir à une configuration récemment visitée.

La stratégie de recherche Tabou, introduite par Glover (Glover, 1989; Glover, 1990), est une méta-heuristique de recherche locale dans laquelle une mémoire, appelée liste Tabou, conserve quelques uns des mouvements récemment effectués. Afin d'éviter les cycles, cette méthode interdit ensuite d'effectuer l'inverse d'un de ces mouvements, sauf si ce mouvement satisfait un certain critère d'aspiration.

Définition Étant donné une instance $(\mathcal{S}, f)$ d'un OC, et un voisinage défini par un ensemble $\mathcal{M}$ de mouvements élémentaires, une mémoire Tabou est une fonction $\Lambda : \mathcal{M} \rightarrow \mathbb{N}^+$ qui associe à chaque mouvement $\mu \in \mathcal{M}$ l'itération $\Lambda(\mu)$ jusqu'à laquelle μ est interdit. Par ailleurs, une rétention Tabou est une fonction $\delta : \mathcal{M} \rightarrow \mathbb{N}^+$ qui associe à chaque mouvement $\mu \in \mathcal{M}$ le nombre d'itérations $\delta(\mu)$ durant lequel le mouvement inverse μ^{-1} est interdit. Finalement, soit l'itération i où un mouvement μ est exécuté, on a $\Lambda(\mu^{-1}) = i + \delta(\mu)$.

En général, un algorithme de recherche Tabou fonctionne comme suit. La mémoire Λ est d'abord initialisée pour que chaque mouvement soit autorisé à l'itération initiale. En commençant avec une configuration s, l'algorithme choisit, à chaque itération i, un mouvement μ autorisé (i.e. $i > \Lambda(\mu)$), minimisant la fonction de coût f. Si tous les mouvements du voisinage sont interdits, μ est choisi aléatoirement. La configuration suivante est ensuite générée en appliquant μ à s, et la mémoire Tabou mise jour pour interdire le mouvement inverse μ^{-1} jusqu'à l'itération $i+\delta(\mu)$. Par ailleurs, un mouvement interdit peut quand même être choisi si ce mouvement satisfait le critère d'aspiration. Le critère d'aspiration permet à certains mou-

vements d'échapper à la restriction Tabou, si ces mouvements sont utiles à la recherche locale. Ainsi, le critère d'aspiration le plus utilisé consiste à autoriser un mouvement qui génère une configuration améliorant le meilleur coût obtenu par la recherche. Finalement, l'algorithme se termine lorsqu'une solution (i.e. une configuration que l'on sait optimale) est trouvée, ou après avoir effectué un certain nombre d'itérations. Par ailleurs, la rétention δ est un paramètre critique pour la recherche Tabou. Ce paramètre contrôle le caractère stochastique de la recherche. Ainsi, si la rétention des mouvements est trop grande, tous les mouvements deviendront interdits après un certain nombre d'itérations, et le parcours de la recherche locale sera purement aléatoire. Par contre, si la rétention est trop courte ou même nulle, l'algorithme choisira toujours le meilleur mouvement, ce qui induira des cycles dans la recherche. Cependant, il n'existe pas de paramètre δ approprié à chaque instance d'un OC. La rétention doit donc être ajustée pour chaque instance particulière.

4.1.2.1 Algorithme pour le MWCSP

L'algorithme 17 est une variante de l'algorithme 14 de recherche locale pour le MWCSP, dans lequel un stratégie Tabou a été ajoutée. Cette stratégie consiste à interdire, pendant un certain nombre d'itérations, de ré-assigner à une variable d'une contrainte en conflit la dernière valeur que cette variable a perdue. On remarque que cet algorithme prend un paramètre de plus, un entier positif τ qui est une constante de rétention Tabou pour tous les mouvements (i.e. $\delta(\mu) = \tau,\ \forall\ \mu \in \mathcal{M}$). Comme pour l'algorithme de recherche locale, cet algorithme initialise d'abord l'itération courante i à 1 et génère une affectation complète a en assignant à chaque variable une valeur de son domaine choisie aléatoirement. Cependant, la mémoire Tabou Λ est fixée à 0 pour chaque assignation, afin d'autoriser ces assignations dès la première itération. Ensuite, pour chaque variable x impliquée dans une

contrainte violée par a et chaque nouvelle valeur v du domaine de cette variable, une affectation voisine a' est obtenue en assignant v à x, si cette assignation est autorisée par la mémoire Tabou (i.e. $i > \Lambda(x, v)$) ou améliore le coût de a^* (i.e. critère d'aspiration). Pour chaque affectation autorisée a', si le coût de a' est inférieur à celui de tous les voisins évalués, B est modifié pour ne contenir que (x, v). Si, par ailleurs, a' est de coût égal, (x, v) est ajouté à B. Après avoir ainsi évalué tous les voisins de a, une assignation (x', v') est choisie aléatoirement dans B, ou parmi toutes les assignation possibles, si B est vide. L'affectation de la prochaine itération est ensuite obtenue en appliquant (x', v') à a, et la mémoire Tabou mise à jour pour interdire de ré-assigner à x' sa valeur courante dans a pendant τ itérations (i.e. $\Lambda(x', a(x')) \leftarrow i + \tau$). Finalement, l'itération courante i est incrémentée de 1, et l'algorithme se termine lorsque i dépasse $i_{\max}$.

Illustrons l'algorithme 17 sur le problème de 2-coloriage du graphe de la figure 3.1. Supposons que le nombre maximum d'itérations soit $i_{\max} = 5$, que le paramètre de rétention Tabou vaille $\tau = 4$, et que l'affectation initiale générée par l'algorithme soit $a_1 = (1, 1, 2, 2, 1, 1)$. L'affectation a_1 viole les contraintes c_1, c_2, c_4, c_5 et c_7. Comme aucune assignation n'est interdite, l'assignation $(x_6, 2)$ de coût -3, est d'abord choisie. Cette assignation donne l'affectation $a_2 = (1, 1, 2, 2, 1, 2)$ qui viole les contraintes c_1 et c_5. De plus, l'assignation inverse $(x_6, 1)$, qui remet à x_6 la valeur qu'elle vient de perdre, est ensuite interdite pour $\tau = 4$ itérations (i.e. jusqu'à l'itération $i = 5$). Les meilleures assignations de l'itération 2, $(x_1, 2)$, $(x_3, 1)$ et $(x_4, 1)$, ont alors un coût nul. Une de ces assignations est ensuite choisie au hasard, par exemple $(x_1, 2)$ qui donne $a_3 = (2, 1, 2, 2, 1, 2)$, et son inverse $(x_1, 1)$ est interdite jusqu'à $i = 6$. Les meilleures assignations de l'itération 3, qui ont aussi un coût nul, sont alors $(x_1, 1)$, $(x_3, 1)$ et $(x_4, 1)$. Il n'est cependant pas permis de choisir $(x_1, 1)$, car cette assignation est interdite par la restriction Tabou. Sans cette restriction, la recherche pourrait revenir à l'affectation précédente en choisissant $(x_1, 1)$. Par

Algorithme 17 Algorithme de recherche Tabou pour le MWCSP

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Une fonction de pondération w;
Un entier $i_{\max} > 0$;
Un entier $\tau \geq 0$;
Sortie : Une affectation complète a^*.

Initialisation
pour tout $x \in \mathcal{X}$ **faire**
  $a(x) \leftarrow \text{random}(D_x)$;
  pour tout $v \in D_x$ **faire**
    $\Lambda(x, v) \leftarrow 0$;
  fin pour
fin pour
$a^* \leftarrow a$;
$i \leftarrow 1$;

Recherche locale
tant que $i \leq i_{\max}$ **faire**
  $T \leftarrow \emptyset$;
  pour tout $c \in U_C(a)$ **faire**
    $T \leftarrow T \cup \mathcal{X}(c)$;
  fin pour
  $B \leftarrow \emptyset$;
  $l \leftarrow 0$;
  pour tout $x \in T$ **faire**
    $a' \leftarrow a$;
    pour tout $v \in D_x$, $v \neq a(x)$ **faire**
      $a'(x) \leftarrow v$;
      si $i > \Lambda(x, v)$ ou $f_C(w, a') < f_C(w, a^*)$ **alors**
        si $f_C(w, a') < l$ ou $B = \emptyset$ **alors**
          $B \leftarrow \{(x, v)\}$;
          $l \leftarrow f_C(w, a')$;
        sinon si $f_C(w, a') = l$ **alors**
          $B \leftarrow B \cup \{(x, v)\}$;
        fin si
      fin si
    fin pour
  fin pour
  si $B \neq \emptyset$ **alors**
    $(x', v') \leftarrow \text{random}(B)$;
  sinon
    $x' \leftarrow \text{random}(U_C(a))$;
    $v' \leftarrow \text{random}(D_{x'} \setminus \{a(x')\})$;
  fin si
  $a(x') \leftarrow v'$;
  $\Lambda(x', a(x')) \leftarrow i + \tau$;
  si $f_C(w, a) < f_C(w, a^*)$ **alors**
    $a^* \leftarrow a$;
  fin si
  $i \leftarrow i + 1$;
fin tant que

ailleurs, supposons que $(x_3, 1)$ soit choisie, l'affectation devient $a_4 = (2, 1, 1, 2, 1, 2)$ et $(x_3, 2)$ est interdit jusqu'à $i = 7$. Les meilleurs assignations de l'itération 4 sont ensuite $(x_1, 1)$ et $(x_3, 2)$, qui ont un coût nul. Comme ces assignations sont toutes les deux interdites, il faut alors choisir une assignation de coût +1, par exemple $(x_2, 2)$ qui donne $a_5 = (2, 2, 1, 2, 1, 2)$, et interdire $(x_2, 2)$ jusqu'à $i = 8$. Finalement, à l'itération 5, la seule assignation de coût -1, $(x_1, 1)$, est interdite. Cependant, cette assignation est quand même choisie, car l'affectation optimale résultante $(1, 2, 1, 2, 1, 2)$ est la meilleure trouvée depuis le début de la recherche[4]. Finalement, comme $i = i_{\max}$, l'algorithme doit s'arrêter à cette itération.

4.1.2.2 Algorithme pour le MPWCSP

L'algorithme 18 est une heuristique de recherche Tabou pour le MPWCSP, qui interdit, pendant un certain nombre d'itérations τ, d'instancier une variable avec la valeur que cette variable a perdue en dernier. Comme pour l'algorithme de recherche locale sans restriction Tabou, cet algorithme génère une affectation a dans laquelle aucune variable n'est instanciée, et initialise l'itération courante i à 1. La mémoire Tabou est également initialisée, afin permettre l'application de n'importe quelle assignation dès la première itération. Ensuite, pour chaque variable x non-instanciée et chaque valeur v du domaine de cette variable, *procAssigne* retourne une affectation légale a' dans laquelle x est instanciée avec la valeur v, si (x, v) est autorisée par la mémoire Tabou, ou améliore le coût de a^*. Pour chaque affectation autorisée a', si le coût de a' est inférieur à celui de tous les voisins évalués, B est modifié pour ne contenir que (x, v). Si, par ailleurs, a' est de coût égal, (x, v) est ajouté à B. Après avoir ainsi évalué tous les voisins de a, une assignation (x', v') est choisie aléatoirement de B, ou parmi toutes les assignation

[4]Cette affectation ne viole que la contrainte c_4.

possibles, si B est vide. L'affectation de la prochaine itération est ensuite obtenue de *procAssigne* avec (x', v'), et la mémoire Tabou mise à jour pour interdire de réinstancier toutes les variables désinstanciées par *procAssigne* (i.e. $U_{\mathcal{X}}(a') \setminus U_{\mathcal{X}}(a)$), avec leur valeur courante dans a pendant τ itérations. Finalement, l'itération courante i est incrémentée de 1, et l'algorithme se termine lorsque i dépasse $i_{\max}$.

Illustrons, une fois de plus, l'algorithme 18 sur le problème de 2-coloriage de la figure 3.1, avec comme paramètres $i_{\max} = 8$ et $\tau = 4$. Supposons que les affectations des quatre premières itérations sont: $a_1 = (-,-,-,-,-,-)$, $a_1 = (1,-,-,-,-,-)$, $a_2 = (1,-,2,-,-,-)$, $a_3 = (1,-,2,-,1,-)$ et $a_4 = (1,-,2,-,1,2)$. Comme aucune variable n'a dû être désinstanciée, lors de ces itérations, toutes les assignations sont autorisées. On remarque que a_4 possède deux variables non-instanciées: x_2 et x_4. Les assignations possibles, à cette itération, sont donc $(x_2, 1)$, $(x_2, 2)$, $(x_4, 1)$ et $(x_4, 2)$. À l'exception de $(x_2, 2)$ qui a un coût de $+1$, ces assignations ont toutes un coût nul, puisque, pour chacune d'elle, une variable doit être désinstanciée. Si on choisit, par exemple, $(x_2, 1)$, on doit alors désinstancier x_1, car c_1 est violée, et interdire l'assignation inverse $(x_1, 1)$ jusqu'à l'itération $i = 8$. L'affectation résultante $a_5 = (-.1,2,-,1,2)$ a x_1 et x_4 non-instanciées, et les meilleures assignations, qui ont un coût nul, sont $(x_1, 1)$, $(x_1, 2)$, $(x_4, 1)$ et $(x_4, 2)$. Comme $(x_1, 1)$ est interdite, on choisit une des autres assignations, par exemple $(x_4, 1)$ qui donne une affectation violant c_7. Il faut donc désinstancier x_5 et interdire $(x_5, 1)$ jusqu'à $i = 9$. Ensuite, l'affectation $a_6 = (-,1,2,1,-,2)$ a quatre assignations optimales de coût nul: $(x_1, 1)$, $(x_1, 2)$, $(x_5, 1)$ et $(x_5, 2)$. Comme $(x_1, 1)$ et $(x_5, 1)$ sont interdites, on peut, par exemple, choisir $(x_1, 2)$ qui donne une affectation violant c_2. Ainsi, il faut désinstancier x_6 et interdire $(x_6, 2)$ jusqu'à $i = 10$. L'affectation $a_7 = (2,1,2,1,-,-)$ a alors une seule assignation de coût -1, $(x_5, 2)$, qui donne l'affectation optimale $a_8 = (2,1,2,1,2,-)$. On remarque, par ailleurs, que même si cette dernière assignation était interdite par la mémoire Tabou, elle

Algorithme 18 Algorithme de recherche Tabou pour le MPWCSP

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$:
Une fonction de pondération w;
Un entier $i_{\max} > 0$;
Un entier $\tau \geq 0$;
Sortie : Une affectation complète a^*.

Initialisation
pour tout $x \in \mathcal{X}$ **faire**
  $a(x) \leftarrow$ aucune valeur;
  pour tout $v \in D_x$ **faire**
    $\Lambda(x, v) \leftarrow 0$;
  fin pour
fin pour
$a^* \leftarrow a$ = affectation vide;
$i \leftarrow 1$;

Recherche locale
tant que $i \leq i_{\max}$ **faire**
  $B \leftarrow \emptyset$;
  $l \leftarrow 0$;
  pour tout $x \in U_{\mathcal{X}}(a)$ **faire**
    pour tout $v \in D_x$, $v \neq a(x)$ **faire**
      $a' \leftarrow$ procAssigne$(P, w, a, (x, v))$;
      si $i > \Lambda(x, v)$ ou $f_{\mathcal{X}}(w, a') < f_{\mathcal{X}}(w, a^*)$ **alors**
        si $f_{\mathcal{X}}(w, a') < l$ ou $B = \emptyset$ **alors**
          $B \leftarrow \{(x, v)\}$;
          $l \leftarrow f_{\mathcal{X}}(w, a')$;
        sinon si $f_{\mathcal{X}}(w, a') = l$ **alors**
          $B \leftarrow B \cup \{(x, v)\}$;
        fin si
      fin si
    fin pour
  fin pour
  si $B \neq \emptyset$ **alors**
    $(x', v') \leftarrow$ random(B);
  sinon
    $x' \leftarrow$ random$(U_{\mathcal{X}}(a))$;
    $v' \leftarrow$ random$(D_{x'} \setminus \{a(x')\})$;
  fin si
  $a' \leftarrow$ procAssigne$(P, w, a, (x', v'))$;
  pour tout $x \in U_{\mathcal{X}}(a') \setminus U_{\mathcal{X}}(a)$ **faire**
    $\Lambda(x, a(x)) \leftarrow i + \tau$;
  fin pour
  $a \leftarrow a'$;
  si $f_{\mathcal{X}}(w, a) < f_{\mathcal{X}}(w, a^*)$ **alors**
    $a^* \leftarrow a$;
  fin si
  $i \leftarrow i + 1$;
fin tant que

serait néanmoins valide puisqu'elle satisfait le critère d'aspiration en donnant la meilleure affectation trouvée. Enfin, l'algorithme doit s'arrêter à cette itération car $i = i_{\max}$.

4.1.3 Détails d'implantation des algorithmes

Un des avantages des CN provient de l'interaction des contraintes, faite au niveau du domaine des variables. La programmation par contraintes est un paradigme de résolution qui propage l'effet d'une contrainte à travers le domaine des variables, afin de réduire celui-ci. Cette section propose une manière efficace d'implanter certaines parties critiques des algorithmes de recherche locale pour le MWCSP et le MWPCSP.

Dans les algorithmes de recherche locale pour le MWCSP et le MPWCSP, l'étape la plus critique est l'évaluation du coût des assignations à une itération. Ainsi, soit a l'affectation de cette itération, le nombre d'assignations à évaluer est

$$\sum_{c \in U_C(a)} \left(\sum_{x \in \mathcal{X}(c)} |D_x| - 1 \right)$$

dans le cas du MWCSP et

$$\sum_{x \in U_{\mathcal{X}}(a)} |D_x|$$

dans le cas du MPWCSP. Il est cependant possible d'évaluer le coût d'une assignation en temps constant à l'aide d'une fonction $\Gamma : \mathcal{M} \rightarrow \mathbb{R}^+$ qui associe un coût positif $\Gamma(x, v)$ à chaque assignation $(x, v) \in \mathcal{M}$. Cette fonction s'implante facilement à l'aide d'une simple table de dimensions $|\mathcal{X}| \times \max |D_x|$. Le coût d'une assignation (x, v) correspond simplement à la différence entre le coût d'assigner v à x et le coût de lui laisser sa valeur courante dans a (i.e. $\Gamma(x, v) - \Gamma(x, a(x))$).

Cette table doit, par ailleurs, être mise à jour après chaque assignation.

L'algorithme 19 montre comment certaines structures de données sont mises à jour, avant d'appliquer une assignation (x, v) à a. Cet algorithme prend le CN incohérent P, la fonction de pondération w, l'affectation courante a, une assignation (x, v), une fonction de coût Γ. Les autres paramètres sont la fonction $\Omega : \mathcal{X} \longrightarrow \mathbb{N}^+$ qui associe à chaque variable $x \in \mathcal{X}$ le nombre de contraintes violées $\Omega(x)$ dans lesquelles cette variable est impliquée, et l'ensemble Q des contraintes violées en assignant (x, v) à a. La fonction Ω sert à l'évaluation des variables à désinstancier dans l'algorithme 15. Ainsi, lorsqu'une variable impliquée dans une contrainte violée doit être désinstanciée, on choisit, parmi celles de plus grand poids, une variable x ayant la plus grande valeur $\Omega(x)$. L'ensemble Q est également important pour les algorithmes de recherche locale. En particulier, Q donne les contraintes violées, dont l'assignation des variables forme le voisinage de l'itération suivante, dans le cas du MWCSP.

Algorithme 19 Algorithme de mise-à-jour des structures de données

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Une fonction de pondération w;
L'affectation courante a;
Une assignation (x, v);
E/S : Une fonction de coût Γ;
Une fonction de conflit Ω;
Un ensemble Q de contraintes.

pour tout $c \in \mathcal{C}(x)$ **faire**
  procContrainte$(c, P, w, a, (x, v), \Gamma, \Omega, Q)$;
fin pour

Le fonctionnement de cet algorithme est simple. Pour chaque contrainte impliquant la variable modifiée par l'assignation, l'algorithme appelle une procédure *procContrainte* qui met à jour Γ, Ω et Q. Chacune de ces contraintes est donc responsable de mettre en partie à jour l'information contenue dans ces structures.

Par ailleurs, la manière dont cette mise-à-jour est faite dépend du type de contrainte, et du problème à résoudre (i.e. MWCSP ou MPWCSP). L'algorithme 20 montre l'implantation de *procContrainte* pour une contrainte d'inégalité[5] dans le cas du MWCSP. Dans cet algorithme, x' représente la variable de la contrainte binaire qui n'est pas dans l'assignation (i.e. $x' \in \mathcal{X}(c)$, $x' \neq x$). Comme c impose à x et x' d'avoir des valeurs différentes, et puisque la valeur de x vient de changer, on enlève d'abord à Γ le coût $w(c)$ d'assigner à x' la valeur précédente de x, et on lui ajoute le coût $w(c)$ d'assigner à x' la nouvelle valeur v de x. Finalement, si c était en conflit, on décrémente le nombre de contraintes violées dans lesquelles sont impliquées x et x', et on retire c de l'ensemble Q. Si, par ailleurs, c est violée suite à l'assignation (x, v), on incrémente $\Omega(x)$ et $\Omega(x')$, et on ajoute c à Q.

Algorithme 20 Mise-à-jour d'une contrainte d'inégalité pour le MWCSP

Entrée: La contrainte c;
Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Une fonction de pondération w;
L'affectation courante a;
Une assignation (x, v);
E/S : Une fonction de coût Γ;
Une fonction de conflit Ω;
Un ensemble Q de contraintes.

$\{x'\} \leftarrow \mathcal{X}(c) \setminus \{x\}$;
$\Gamma(x', a(x)) \leftarrow \Gamma(x', a(x)) - w(c)$;
$\Gamma(x', v) \leftarrow \Gamma(x', v) + w(c)$;
si $a(x') = a(x)$ **alors**
  $\Omega(x') \leftarrow \Omega(x') - 1$;
  $\Omega(x) \leftarrow \Omega(x) - 1$;
  $Q \leftarrow Q \setminus \{c\}$;
sinon si $a(x') = v$ **alors**
  $\Omega(x') \leftarrow \Omega(x') + 1$;
  $\Omega(x) \leftarrow \Omega(x) + 1$;
  $Q \leftarrow Q \cup \{c\}$;
fin si

L'algorithme 21 montre l'implantation pour les contraintes d'inégalité, dans le cas

[5]Le type de contrainte utilisé dans la modélisation d'un problème de k-coloriage de graphe.

du MPWCSP. Contrairement au MWCSP, il faut tenir compte de l'instanciation de x et x'. Il peut ainsi y avoir deux cas. Dans le premier cas, x n'est pas instanciée, mais le devient avec l'assignation (x, v). L'instanciation de x entraîne la possibilité d'un conflit si on assigne à x' la même valeur. Comme ce conflit entraîne x à être désinstanciée, il faut donc ajouter à Γ le coût $w(x)$ d'assigner à x' la valeur v. Par ailleurs, si x et x' ont, suite à l'assignation, la même valeur, il faut incrémenter Ω pour ces deux variables et rajouter c à Q. Dans l'autre cas, x est instanciée, mais doit est désinstanciée suite à l'assignation. Comme x ne sera pas instanciée, on peut assigner à x' n'importe quelle valeur sans créer de conflit. Il faut alors enlever de Γ le coût d'assigner à x' l'ancienne valeur de x. Finalement, si c était en conflit, il faut décrémenter Ω pour x et x' et retirer c de Q.

Algorithme 21 Mise-à-jour d'une contrainte d'inégalité pour le MPWCSP

Entrée: La contrainte c;
Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Une fonction de pondération w;
L'affectation courante a;
Une assignation (x, v);
E/S : Une fonction de coût Γ;
Une fonction de conflit Ω;
Un ensemble Q de contraintes.

$\{x'\} \leftarrow \mathcal{X}(c) \setminus \{x\}$;
si $v \neq$ "désinstancie x" **alors**
  $\Gamma(x', v) \leftarrow \Gamma(x', v) + w(x)$;
  si $a(x') = v$ **alors**
    $\Omega(x') \leftarrow \Omega(x') + 1$;
    $\Omega(x) \leftarrow \Omega(x) + 1$;
    $Q \leftarrow Q \cup \{c\}$;
  fin si
sinon
  $\Gamma(x', a(x)) \leftarrow \Gamma(x', a(x)) - w(x)$;
  si $a(x') = a(x)$ **alors**
    $\Omega(x') \leftarrow \Omega(x') - 1$;
    $\Omega(x) \leftarrow \Omega(x) - 1$;
    $Q \leftarrow Q \setminus \{c\}$;
  fin si
fin si

4.2 Algorithmes heuristiques de détection d'IIS

Dans la première partie de ce chapitre, il a été montré comment des méta-heuristiques pouvaient être implantées pour le MWCSP et MPWCSP. Cette section décrit comment une version heuristique des algorithmes de retrait, d'insertion et de *hitting set*, ainsi que d'autres techniques aidant à la détection d'IIS, peut être obtenue, à l'aide de ces méta-heuristiques.

4.2.1 Algorithme heuristique de retrait

L'algorithme de retrait, présenté à la section 3.2.1, utilise une procédure *procMWIC* qui résoud de manière exacte le MWCSP, dans le cas de la détection d'IIS de contraintes, et le MPWCSP, dans le cas de la détection d'IIS de variables. On peut, cependant, se demander ce qui se produirait si *procMWIC* était une méta-heuristique, comme celles proposées dans les sections précédentes. Lorsque cet algorithme retire une contrainte ou une variable de P, faisant partie de l'intersection des IIS, P devient cohérent, et *procMWIC* obtient une affectation a de coût $f(w, a) = 0$. Cette contrainte ou variable est alors ré-insérée. Toutefois, si *procMWIC* n'est pas exacte, cette procédure peut retourner une affectation sous-optimale (i.e. $f(w, a) > 0$) de telle sorte que la contrainte ou variable n'est pas ré-insérée. En conséquence, l'algorithme retourne un ensemble K qui est cohérent. Considérons, par exemple, la détection d'un IIS de contraintes pour le problème de 2-coloriage de la figure 3.1. Supposons que c_4 soit d'abord retirée, le CN devient cohérent. Cependant, si *procMWIC* retourne une affectation sous optimale, par exemple $(1, 2, 1, 2, 1, 1)$, c_4 ne sera pas ré-insérée, et l'ensemble K sera nécessairement cohérent. L'algorithme heuristique de retrait possède ainsi la propriété suivante:

Propriété 4.2.1 *Soit P un CN incohérent. L'algorithme heuristique de retrait*

produit, en un nombre fini d'itérations, un ensemble K de contraintes ou de variables qui est soit un IIS, ou est cohérent.

Preuve Supposons que K soit un ensemble incohérent. Soit e_i n'importe quelle contrainte ou variable de K retirée à l'itération i, et soit K_i l'ensemble des contraintes ou variables ayant un poids de 1 a l'itération i. On sait que $K_i \setminus \{e_i\}$ est cohérent. Par ailleurs, comme $K \subseteq K_i$, l'ensemble $K \setminus \{e_i\}$ est également cohérent. L'ensemble K est donc un IIS.

4.2.2 Algorithme heuristique d'insertion

Il est également possible d'obtenir un version heuristique de l'algorithme d'insertion en utilisant une méta-heuristique pour *procMWIC*. Tout comme l'algorithme heuristique de retrait, l'algorithme heuristique d'insertion ne garantit pas que l'ensemble K de contraintes ou de variables obtenu soit un IIS. Ainsi, si *procMWIC* obtient une affectation sous-optimale contenant une contrainte ou une variable faisant partie de l'intersection des IIS, cette contrainte ou variable peut être retirée, rendant K cohérent. Soit, par exemple, la détection d'IIS de contraintes dans le problème de 2-coloriage de la figure 3.1. Si *procMWIC* retourne l'affectation sous-optimale $(1, 2, 1, 1, 2, 2)$ violant c_4, c_5 et c_7, l'algorithme conserve une de ces contraintes et retire les autres. Si c_4, qui fait partie de l'intersection des IIS, n'est pas conservée, P sera alors cohérent. Dans un tel cas, l'algorithme peut ne jamais se terminer, s'il est impossible de former un ensemble K incohérent.

L'algorithme 28 illustre comment l'algorithme d'insertion peut être modifié pour tenter de déceler ce type d'erreur. Ainsi, après avoir obtenu une affectation a de *procMWIC*, on vérifie si P est cohérent. Si le coût de a est nul, P est cohérent et l'algorithme retourne un échec. Sinon, l'algorithme poursuit sa détection d'un IIS.

Algorithme 22 Algorithme d'insertion (*modifié*)

```
...
si f(w, a) = 0 alors
  ÉCHEC. Interrompre l'algorithme;
sinon si f(w, a) ≥ |E| alors
  ...
fin si
...
```

Il se peut, cependant, que même avec cette modification, l'algorithme ne parvienne pas à déceler l'erreur, si *procMWIC* est incapable d'obtenir une affectation optimale. Dans ce cas, l'algorithme produit un ensemble K de contraintes ou de variables qui est cohérent. Par ailleurs, il est également possible que l'algorithme s'arrête prématurément, si *procMWIC* obtient une affectation sous-optimale a de coût $f(w, a) \geq |\mathcal{E}|$. Alors que l'algorithme croit avoir un ensemble K incohérent, cet ensemble peut, en réalité, être cohérent. Enfin, l'algorithme heuristique d'insertion possède la propriété suivante:

Propriété 4.2.2 *Soit P un CN incohérent. L'algorithme heuristique d'insertion produit, en un nombre fini d'itérations, un ensemble K de contraintes ou de variables qui est soit un IIS, ou est cohérent.*

Preuve Supposons que K soit un ensemble incohérent, a_i l'affectation optimale obtenue à l'itération i, et $e_i \in U(a_i)$ n'importe quelle contrainte ou variable de K conservée à cette itération. On sait que $\mathcal{E} \setminus U(a_i)$ est un ensemble cohérent. De plus, comme $K \setminus \{e_i\} \subseteq \mathcal{E} \setminus U(a_i)$, l'ensemble $K \setminus \{e_i\}$ est aussi cohérent. L'ensemble K est donc un IIS.

4.2.3 Algorithme heuristique de *hitting set*

Tout comme les algorithmes heuristiques de retrait et d'insertion, la version heuristique de l'algorithme de *hitting set* n'offre aucune garantie que l'ensemble K obtenu soit un IIS. Ainsi, *procMWIC* peut obtenir une affectation sous-optimale a de coût $f(w, a) \geq |\mathcal{E}|$ et se terminer avec un ensemble K cohérent. Par ailleurs, si cet algorithme retourne un ensemble K incohérent, K est un IIS uniquement si *procMHS* obtient des HS minimaux au sens de l'inclusion. De même, K est un IIS minimum seulement si *procMHS* obtient des HS de cardinalité minimale. L'algorithme heuristique de *hitting set* possède donc la propriété suivante:

Propriété 4.2.3 *Soit P un CN incohérent. Si* procMHS *retourne des HS minimaux au sens de l'inclusion, l'algorithme heuristique par* hitting set *produit, en un nombre fini d'itérations, un ensemble K de contraintes ou de variables qui est soit un IIS, ou est cohérent. Par ailleurs, si* procMHS *retourne des HS de cardinalité minimale, K est soit un IIS minimum, ou est cohérent.*

Preuve Supposons que l'algorithme se termine avec un ensemble K incohérent, et soit e une contrainte ou une variable de K. Si *procMHS* retourne des HS minimaux au sens de l'inclusion, il existe nécessairement un ensemble $U(a_i)$, obtenu à une itération i quelconque, qui n'intersecte pas avec l'ensemble $K \setminus \{e\}$. Par ailleurs, on sait que $\mathcal{E} \setminus U(a_i)$ est un ensemble cohérent. Ainsi, comme $K \setminus \{e\} \subset \mathcal{E} \setminus U(a_i)$, l'ensemble $K \setminus \{e\}$ est également cohérent. L'ensemble K est donc un IIS. Finalement, si *procMHS* retourne des HS de cardinalité minimale, K est nécessairement un IIS minimum.

4.3 Stratégies de récupération d'erreur

Lorsque l'instance à résoudre n'est pas trop grande, et si les paramètres de la méta-heuristique *procMWIC* sont bien ajustés pour cette instance, les affectations obtenues par *procMWIC* sont souvent optimales. Il arrive, cependant, que ces affectations soient sous-optimales pour certaines instances de taille moyenne. Enfin, pour des instances de plus grande taille, il est rare que *procMWIC* trouve des affectations optimales. Par ailleurs, il a été montré que la capacité des algorithmes heuristiques de détection à obtenir un IIS dépendait justement de l'optimalité de ces affectations[6]. Cette section présente quelques stratégies permettant de traiter certaines erreurs de détection causées par l'obtention d'affectations sous-optimales.

4.3.1 Algorithme de réduction

Pour une instance où la méta-heuristique est susceptible de ne pas trouver d'affectations optimales, il est préférable d'obtenir un ensemble incohérent K réduit, même si K n'est pas un IIS, que d'avoir K cohérent (i.e. échec de détection). On peut ainsi ré-appliquer un algorithme de détection sur le CN réduit, pour lequel la méta-heuristique aura de meilleures chances de trouver des affectations optimales. L'algorithme 23 décrit une procédure générale pour réduire un CN. Cet algorithme prend en entrée un CN possiblement incohérent P, une paire d'entiers $0 < i^{(1)}_{\max} \leq i^{(2)}_{\max}$ ainsi qu'un entier positif τ, et retourne un ensemble possiblement incohérent de contraintes ou de variables K. Les entiers $i^{(1)}_{\max}$ et $i^{(2)}_{\max}$ correspondent à deux "réglages" de la méta-heuristique, tel que $i^{(2)}_{\max}$ alloue un grand nombre égal ou supérieur d'itérations que $i^{(1)}_{\max}$. Ainsi, la méta-heuristique a plus de chances d'obtenir une affectation optimale avec $i^{(2)}_{\max}$ qu'avec $i^{(1)}_{\max}$. De plus, τ correspond au

[6] L'algorithme de *hitting set* doit également obtenir de *procMHS* des HS de cardinalité minimale ou minimum.

paramètre de rétention de mémoire Tabou de la méta-heuristique. Cet algorithme utilise, par ailleurs, quatre nouvelles procédures. Tout d'abord, *procMWIC-2* est une procédure similaire à *procMWIC* qui prend comme paramètre supplémentaire l'affectation initiale de la recherche. Ainsi, *procMWIC-2* retourne une affectation dont le coût est égal ou inférieur à celui de l'affectation en entrée. Ensuite, *procInitialise*, *procModifie* et *procRépare* sont des procédures qui implantent les fonctions d'*initialisation*, de *modification*, et de *réparation* d'un algorithme de détection quelconque.

Cet algorithme initialise d'abord le poids des contraintes ou variables de $\mathcal{E}$ à 1, de même que l'ensemble K pour qu'il soit identique à $\mathcal{E}$, et appelle *procInitialise* qui initialise la détection. Ensuite, une affectation a est obtenue de *procMWIC* en $i_{\max}^{(1)}$ itérations. Si $f(w,a) \geq |\mathcal{E}|$, les contraintes ou variables de poids $|\mathcal{E}|$ forment un ensemble potentiellement incohérent qu'il faut valider. Cette validation détermine, à elle seule, si K sera cohérent ou non. Ainsi, pour s'assurer qu'il n'existe aucune affectation de coût inférieur à $|\mathcal{E}|$, deux techniques sont utilisées. La première consiste à "extraire" K en utilisant une nouvelle pondération w' pour laquelle les poids de $|\mathcal{E}|$ sont fixés à 1, et les autres à 0. Cette technique aide *procMWIC* en réduisant le nombre de contraintes ou de variables de P, ainsi qu'en n'ayant qu'une seule valeur non nulle pour les poids[7]. La seconde stratégie consiste à donner, lors de la validation, plus de temps à la méta-heuristique pour obtenir une affectation optimale, en lui passant $i_{\max}^{(2)}$ en paramètre. Une affectation a' est ainsi obtenue par *procMWIC*. Si le coût de a' est supérieur à 0, l'ensemble K des contraintes ou variables de poids non nul est possiblement incohérent, et l'algorithme retourne cet ensemble. Sinon, a' est une affectation telle $f(w,a') < |\mathcal{E}|$ [8]. On peut ainsi utiliser a' comme affectation initiale de la recherche donnant l'affectation a de

[7]Des expériences ont montré que la recherche locale était handicapée par l'utilisation de deux valeurs non nulles de poids.

[8]À noter qu'on utilise w et non w'.

l'itération suivante. L'étape de validation qui vient d'être décrite correspond au cas où $f(w,a) \geq |\mathcal{E}|$. Si, par ailleurs, $0 < f(w,a) < |\mathcal{E}|$, P est possiblement toujours incohérent, et la procédure *procModifie* est appelée. Ensuite, l'affectation a de la prochaine itération est obtenue de *procMWIC* en $i_{\max}^{(1)}$ itérations. Finalement, dans le dernier cas, si le coût $f(w,a) = 0$, P est devenu cohérent suite au retrait de contraintes ou de variables[9]. L'algorithme appelle alors *procRépare*, qui selon son implantation, rend P à nouveau incohérent en ré-insérant une ou plusieurs contraintes ou variables. Par contre, si *procRépare* est incapable de rendre P à nouveau incohérent, l'algorithme se termine et retourne l'ensemble $\mathcal{E}$ initial.

4.3.1.1 Implantation de retrait

La détection d'un ensemble incohérent dans l'algorithme 23 dépend de l'implantation des procédures *procInitialise*, *procModifie* et *procRépare*. Dans le cas de la détection par *retrait*, l'implantation de ces procédures est basée sur l'algorithme hybride, présenté à la section 3.3.1. En ce sens, l'ensemble K est représenté par les contraintes ou variables de poids $|\mathcal{E}|$. L'implantation de *procInitialise* vide simplement un ensemble R, contenant l'ensemble des contraintes ou variables retirées, ainsi qu'un ensemble L contenant la dernière contrainte ou variable retirée. L'algorithme 25 montre l'implantation de *procModifie*. Comme pour l'algorithme hybride, une contrainte ou variable e de poids 1 est retirée en fixant son poids à 0. Cependant, e est également ajoutée à R et l'ensemble L est modifié pour ne contenir que e. Par ailleurs, l'algorithme 26 donne l'implantation de *procRépare*. Si L contient une contrainte ou variable e retirée à la dernière itération, e fait probablement partie de l'intersection des IIS, et son poids est fixé à $|\mathcal{E}|$. De plus, comme e a été ré-insérée, on la retire de L. Par contre, si L est vide, la dernière itération

[9]Il est également possible que P était initialement cohérent.

Algorithme 23 Algorithme général de réduction

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Une fonction de pondération w;
Une paire d'entiers $0 < i^{(1)}_{\max} \leq i^{(2)}_{\max}$;
Un entiers $\tau \geq 0$;
Sortie : Un ensemble de contraintes ou de variables K.

Initialisation
pour tout $e \in \mathcal{E}$ **faire**
  $w(e) \leftarrow 1$;
fin pour
$K \leftarrow \mathcal{E}$;
procInitialise():
$a \leftarrow$ procMWIC$(P, w, i^{(1)}_{\max}, \tau)$;
Continuer $\leftarrow$ VRAI;

Construction
tant que Continuer = VRAI **faire**
  si $f(w, a) \geq |\mathcal{E}|$ **alors**
    $w' \leftarrow w$;
    pour tout $e \in \mathcal{E}$ **faire**
      si $w(e) = |\mathcal{E}|$ **alors**
        $w'(e) \leftarrow 1$;
      sinon
        $w'(e) \leftarrow 0$;
      fin si
    fin pour
    $a' \leftarrow$ procMWIC$(\mathcal{P}, w', i^{(2)}_{\max}, \tau)$;
    si $f(w, a') > 0$ **alors**
      $K \leftarrow \{e \mid w'(e) = 1\}$;
      Continuer $\leftarrow$ FAUX;
    sinon
      $a \leftarrow$ procMWIC-2$(P, w, i^{(1)}_{\max}, \tau, a')$;
    fin si
  sinon si $f(w, a) > 0$ **alors**
    procModifie$(P, w, U(a))$;
    $a \leftarrow$ procMWIC$(P, w, i^{(1)}_{\max}, \tau)$;
  sinon
    Continuer $\leftarrow$ procRépare(P, w);
    si Continuer = VRAI **alors**
      $a \leftarrow$ procMWIC$(P, w, i^{(1)}_{\max}, \tau)$;
    fin si
  fin si
fin tant que

a été passée à rendre P incohérent. Cependant, puisque P est toujours cohérent, on sait qu'une erreur a été commise à une itération antérieure (i.e. *procMWIC* n'a pas obtenu d'affectation de coût nul, alors que P était cohérent). Il faut alors ré-insérer une contrainte ou variable précédemment retirée. Ainsi, une contrainte ou variable e' est enlevée de R, et comme e' ne fait pas nécessairement partie de l'intersection des IIS, son poids est fixé à 1. Finalement, si R est vide, P était initialement cohérent, et il est impossible de récupérer.

Algorithme 24 Implantation de *procInitialise* pour l'algorithme de retrait

$R \leftarrow \emptyset$;
$L \leftarrow \emptyset$;

Algorithme 25 Implantation de *procModifie* pour l'algorithme de retrait

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
E/S : Une fonction de pondération w.

Choisir $e \in \mathcal{E}$ tel que $w(e) = 1$;
$w(e) \leftarrow 0$;
$R \leftarrow R \cup \{e\}$;
$L \leftarrow \{e\}$;

On remarque que, malgré les erreurs potentielles, cette implantation permet à l'algorithme 23 de se terminer en un nombre fini d'itérations. Ainsi, à chaque fois qu'un appel à *procModifie* est suivi d'un appel à *procRépare*, le poids d'une contrainte ou d'une variable est fixé à $|\mathcal{E}|$, après quoi il n'est plus possible de modifier ce poids. Puisqu'il est impossible d'avoir une séquence infinie d'appel à *procModifie* (car sinon toutes les contraintes ou variables de $\mathcal{E}$ auraient un poids nul, et P serait trivialement cohérent), ou d'avoir une séquence infinie d'appel à *procRépare* (car R serait vide et l'algorithme se terminerait), le nombre de contraintes ou de variables de poids $|\mathcal{E}|$ doit nécessairement augmenter. Par ailleurs, si K est incohérent, il n'est pas forcément un IIS, comme le montre l'exemple suivant. Considérons la détection d'un IIS de contraintes à l'aide de cet algorithme, sur le problème de

Algorithme 26 Implantation de *procRépare* pour l'algorithme de retrait

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
E/S : Une fonction de pondération w;
Sortie : Une valeur de succès (VRAI ou FAUX).

succès $\leftarrow$ VRAI;
si $L \neq \emptyset$ **alors**
  $\{e\} \leftarrow L$;
  $w(e) \leftarrow |\mathcal{E}|$;
  $L \leftarrow \emptyset$;
  $R \leftarrow R \setminus \{e\}$;
sinon
  si $R \neq \emptyset$ **alors**
    Choisir un élément $e' \in R$;
    $R \leftarrow R \setminus \{e'\}$;
    $w(e') \leftarrow 1$;
  sinon
    succès $\leftarrow$ FAUX;
  fin si
fin si

2-coloriage de la figure 3.1. Supposons que c_4 soit retirée en premier, rendant P cohérent. Si *procMWIC* retourne une affectation sous-optimale, une autre contrainte sera retirée, par exemple c_1, au lieu de que c_4 soit ré-insérée. Si l'algorithme se rend alors compte que P est cohérent, c_1 sera ré-insérée avec un poids de $|\mathcal{E}|$. Encore une fois, il se peut que l'algorithme ne détecte pas que P soit cohérent, et retire une nouvelle contrainte, qui sera ré-insérée à l'itération suivante. Ce processus peut se poursuivre jusqu'à ce que le poids de toutes les contraintes, sauf c_4, soit fixé à $|\mathcal{E}|$. Enfin, après avoir testé toutes les contraintes, l'algorithme peut détecter que P est cohérent, et se rendre compte qu'il a commis une erreur. La seule contrainte de R, c_4, est alors ré-insérée avec un poids de 1, et celle-ci sera retirée puis éventuellement ré-insérée avec un un poids de $|\mathcal{E}|$. L'algorithme retournera, dans ce cas, un ensemble K contenant toutes les contraintes de P.

4.3.1.2 Implantation d'insertion

La récupération d'erreur, dans le cas de l'algorithme d'insertion, est semblable à celle de l'algorithme de retrait. L'implantation de *procInitialise* vide d'abord l'ensemble R des contraintes ou variables retirées. L'algorithme 28 montre ensuite l'implantation de *procModifie*. À chaque itération où P est incohérent, cet algorithme choisit une contrainte ou variable de U et fixe son poids à $|\mathcal{E}|$. De plus, les autres contraintes ou variables de U sont retirées, en fixant leur poids à 0, puis ajoutées à R. L'algorithme 29 montre ensuite l'implantation de *procRépare*, appelée lorsque P devient cohérent. Cet algorithme choisit une contrainte ou une variable de R et la ré-insère en fixant son poids de 1, afin de rendre P à nouveau incohérent.

Algorithme 27 Implantation de *procInitialise* pour l'algorithme d'insertion

$R \leftarrow \emptyset$;

Algorithme 28 Implantation de *procModifie* pour l'algorithme d'insertion

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Entrée: Un ensemble de contraintes ou de variables U;
E/S : Une fonction de pondération w.

si $U \neq \emptyset$ **alors**
  Choisir un élément $e \in U$;
  $w(e) \leftarrow |\mathcal{E}|$;
  pour tout $e' \in U$, $e' \neq e$ **faire**
    $R \leftarrow R \cup \{e'\}$;
    $w(e') \leftarrow 0$;
  fin pour
fin si

Cette implantation permet également à l'algorithme 23 de se terminer en un nombre fini d'itérations. Ainsi, à chaque fois que *procModifie* est appelée, le poids d'une contrainte ou une variable supplémentaire est fixé à $|\mathcal{E}|$. Il est ensuite impossible de re-modifier ce poids. Finalement, puisqu'il est impossible d'avoir une

Algorithme 29 Implantation de *procRépare* pour l'algorithme d'insertion

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
E/S : Une fonction de pondération w;
Sortie : Une valeur de succès (VRAI ou FAUX).

```
succès ← VRAI;
si R ≠ ∅ alors
  Choisir une élément e' ∈ R;
  R ← R \ {e'};
  w(e') ← 1;
sinon
  succès ← FAUX;
fin si
```

séquence infinie d'appel de *procRépare*, l'algorithme se terminera lorsque l'ensemble des contraintes ou variables de poids $|\mathcal{E}|$ sera incohérent. Par ailleurs, l'algorithme n'offre aucune garantie que K soit un IIS. Ainsi, revenant à la détection d'IIS de contraintes sur le problème de 2-coloriage de la figure 3.1, si *procMWIC* retourne, par exemple, l'affectation sous-optimale $a_1 = (1, 1, 2, 1, 2, 1)$, les contraintes violées seront c_1, c_2 et c_4. Le poids d'une contrainte, par exemple c_1, est alors fixé à $|\mathcal{E}|$, par exemple c_1, et celui des autres à 0, de telle sorte que P devienne cohérent. Ensuite, supposons que *procMWIC* retourne une autre affectation sous-optimale $a_2 = (1, 2, 2, 1, 2, 1)$, violant c_3. Le poids de c_3 est alors fixé à $|\mathcal{E}|$. On constate alors que P est toujours cohérent. Cependant, comme c_1 et c_3, chacune ayant un poids de $|\mathcal{E}|$, font partie de deux IIS différents, il est alors impossible que K soit un IIS, même si les récupérations rendent P à nouveau incohérent.

4.3.1.3 Implantation de *hitting set*

Contrairement aux algorithmes de retrait et d'insertion, l'algorithme par hitting set ne possède pas de stratégie de récupération d'erreur, puisque les contraintes ou variables ne sont jamais retirées. Ainsi, le poids des contraintes ou des variables de $\mathcal{E}$

vaut soit 1 ou $|\mathcal{E}|$, mais jamais 0. Il est cependant possible d'implanter *procInitialise* et *procModifie* pour cet algorithme. L'implantation de *procInitialise* vide tout simplement la collection $\mathcal{U}$. Finalement, l'algorithme 31 montre l'implantation de *procModifie*.

Algorithme 30 Implantation de *procInitialise* pour l'algorithme de *hitting set*

$\mathcal{U} \leftarrow \emptyset$;

Algorithme 31 Implantation de *procModifie* pour l'algorithme de *hitting set*

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Entrée: Un ensemble de contraintes ou de variables U;
E/S : Une fonction de pondération w.

$\mathcal{U} \leftarrow \mathcal{U} \cup \{U\}$;
$H \leftarrow$ procMHS($\mathcal{U}$);
pour tout $e \in \mathcal{E}$ **faire**
  si $e \in H$ **alors**
    $w(e) \leftarrow |\mathcal{E}|$;
  sinon
    $w(e) \leftarrow 1$;
  fin si
fin pour

4.3.2 Techniques complémentaires

Les stratégies de récupération d'erreur permettent de déceler et corriger certaines erreurs, provenant de l'utilisation d'algorithmes heuristiques pour le MWCSP et le MPWCSP. Cependant, même avec une stratégie de récupération d'erreur, l'ensemble K peut être cohérent. Il est donc important d'utiliser certaines techniques permettant de minimiser le nombre d'erreurs lors de la détection. Une de ces techniques est d'utiliser une heuristique qui détermine quelle contrainte ou variable est ré-insérée durant une récupération (i.e. dans *procRépare*). Une possibilité est de d'abord réinsérer une contrainte ou variable ayant le plus grand poids de voisinage, car celle-ci est probablement la plus près de l'IIS en construction. Une autre possibilité, se

basant sur le fait qu'une erreur a plus de chances d'avoir été commise lors d'une itération récente à sa détection, est de ré-insérer les contraintes ou variables selon l'ordre inverse de leur retrait. D'autres techniques peuvent être employées pour minimiser les erreurs lors de la détection. Une première technique consiste à éviter le retrait ou la ré-insertion répétitive de la même contrainte ou variable, si cette opération mène plusieurs fois à une erreur. Enfin, une autre technique consiste à augmenter la robustesse de la méta-heuristique, par exemple d'utiliser $i_{\max}^{(2)}$ au lieu de $i_{\max}^{(1)}$, lorsque le nombre d'erreurs consécutives dépasse un certain seuil, et la diminuer si le nombre d'erreurs redescend en dessous de ce seuil.

Par ailleurs, si K est incohérent, cet ensemble n'est pas nécessairement un IIS. Ainsi, une contrainte ou variable dont le poids a été fixé à $|\mathcal{E}|$ alors que P était cohérent, ne fera pas forcément partie de l'IIS. On peut cependant utiliser des techniques pour augmenter les chances que K soit un IIS. Une première technique consiste à d'abord appliquer l'algorithme 23 sur P, et ensuite lui appliquer un algorithme exact de détection d'IIS. Le fondement de cette technique est que l'algorithme 23 permet de simplifier P en lui réduisant le nombre de variables et de contraintes. L'algorithme exact de détection, qui, vu sa complexité, ne pouvait pas être utilisé sur le CN original, a alors de meilleures chances d'obtenir un IIS du problème réduit. Une variante de cette technique est d'appliquer un algorithme heuristique de détection qui garantit l'obtention d'un IIS K, si K est incohérent. Encore une fois, cette technique repose sur l'idée que l'algorithme heuristique de détection, qui aurait échoué sur le CN original, est plus apte à réussir sur le CN réduit.

La seconde technique, qui garantit aussi l'obtention d'un IIS, se base sur le fait que *procMWIC* retourne une affectation a dont le coût $f(w, a)$ est une borne supérieure du coût d'une affectation optimale a^*. Ainsi, si $f(w, a) = 1$, le coût de a^* est soit 1 ou 0. En supposant que P est incohérent (i.e. $f(w, a^*) > 0$),

a est alors nécessairement optimale. Par ailleurs, la contrainte ou variable de $U(a)$ fait forcément partie de l'intersection des IIS de P. Ainsi, si on obtient, avec *procMWIC*, $|K|$ affectations dont l'ensemble $U(a)$ contient une variable ou contrainte différente pour chaque affectation, on sait que, si K est incohérent, il forme un IIS. L'algorithme par insertion permet justement de trouver ces affectations. Ainsi, à chaque itération pour laquelle *procMWIC* retourne une affectation a de coût 1, le poids de l'unique contrainte ou variable de $U(a)$ est fixé à $|\mathcal{E}|$. L'ensemble $U(a)$ de l'itération suivante doit donc contenir une contrainte ou une variable différente. En somme, il suffit de vérifier que les affectations obtenues à chaque itération (sauf celles où $f(w, a) \geq |\mathcal{E}|$) ont un coût de 1. Si c'est le cas, et si K est incohérent, cet ensemble est un IIS.

La dernière technique, par ailleurs, ne garantit pas d'obtenir un IIS. Il s'agit d'appliquer itérativement l'algorithme de réduction 23 en utilisant en entrée le résultat de la détection précédente, jusqu'à ce qu'il n'y ait pas de réduction pour un certain nombre d'itérations. L'algorithme 32 illustre cette technique. Cet algorithme prend comme paramètre supplémentaire un entier positif $r_{\max}$ qui représente le nombre maximum d'itérations consécutives autorisées sans réduction, et utilise une procédure *procRéduit* correspondant à l'algorithme 23. Le principe de cet algorithme est que, si on ne parvient pas à réduire K, alors cet ensemble est probablement un IIS. On peut alors augmenter la probabilité que K soit un IIS en relançant la détection à répétition, même si cela n'apporte aucune réduction supplémentaire. Soit α la probabilité que *procRéduit* ne réduise pas un ensemble K incohérent qui n'est pas un IIS, la probabilité que K ne soit pas un IIS, après $r_{\max}$ itérations sans réduction, est égal à $(1 - \alpha)^{r_{\max}}$. On remarque que, si $\alpha < 1$ (i.e. *procRéduit* n'est pas entièrement inefficace), cette probabilité tend vers 0 lorsque $r_{\max}$ augmente. Cependant, comme chaque appel à *procRéduit* prend du temps, il est nécessaire d'utiliser une valeur de $r_{\max}$ qui offre un bon compromis entre le temps de calcul

et la probabilité d'avoir un IIS.

Algorithme 32 Algorithme de réductions successives

Entrée: Un CN incohérent $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$;
Une fonction de pondération w;
Une paire d'entiers $0 < i^{(1)}_{\max} \leq i^{(2)}_{\max}$;
Un entiers $\tau \geq 0$;
Un entier $r_{\max} \geq 0$;
Sortie : Un ensemble de contraintes ou de variables K.

```
K ← E;
r ← 0;
tant que r < r_max faire
  E ← procRéduit( P, w, i^(1)_max, i^(2)_max, τ);
  si |E| < |K| alors
    K ← E;
    r ← 0;
  sinon
    r ← r + 1;
  fin si
fin tant que
```

4.4 Technique d'accélération de détection

Dans la section précédente, il a été montré que, lorsque *procMWIC* retourne une affectation a de coût $f(w, a) = 1$, P est soit cohérent, ou sinon l'unique contrainte ou variable e de $U(a)$ fait partie de l'intersection des IIS. En supposant que P soit incohérent, on peut donc fixer le poids de e à $|\mathcal{E}|$ puisqu'elle fera nécessairement partie de l'IIS contenu dans K. Il est possible d'utiliser ce principe pour accélérer la détection d'IIS. Ainsi, à chaque fois que *procMWIC*, qui peut visiter un très grand nombre d'affectations en un temps très court, visite une affectation a de coût 1, on ajoute l'unique contrainte ou variable de $U(a)$ dans un ensemble T initialement vide. Lorsque *procMWIC* se termine, l'ensemble T contient des contraintes ou des variables qui font forcément partie de K. L'algorithme 33 illustre comment l'algorithme 23 peut être modifié pour bénéficier de cette technique. On remarque

que *procMWIC* retourne en plus d'une affectation a, l'ensemble T de contraintes ou de variables qui vient d'être décrit. À chaque itération, le poids des contraintes ou variables de T est fixé à $|\mathcal{E}|$.

Algorithme 33 Accélération de l'algorithme de réduction

```
...
(a, T) ← procMWIC(P, w, i_max^(1), τ);
Continuer = VRAI;

Construction
tant que Continuer = VRAI faire
  pour tout e ∈ T faire
    w(e) ← |E|;
  fin pour
  si f(w, a) ≥ |E| alors
    ...
    si f(w, a') > 0 alors
      ...
    sinon
      (a, T) ← procMWIC-2(P, w, i_max^(1), τ, a');
    fin si
  sinon si f(w, a) > 0 alors
    ...
    (a, T) ← procMWIC(P, w, i_max^(1), τ);
  sinon
    ...
    si Continuer = VRAI alors
      (a, T) ← procMWIC(P, w, i_max^(1), τ);
    fin si
  fin si
fin tant que
```

L'exemple suivant montre un cas où l'algorithme 33 obtient un IIS en une seule itération. Considérons la détection d'un IIS de contraintes dans le problème de 2-coloriage de la figure 4.2. On remarque que le CN de cette figure est lui-même un IIS de contraintes. Supposons, que le poids des contraintes soit initialisé à 1 et qu'on utilise les paramètres suivant pour *procMWIC*: $i_{\max} = 5$ et $\tau = 5$. On peut vérifier que les affectations suivantes correspondent à une séquence valide

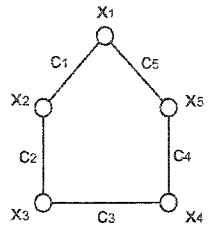

Figure 4.2 Une instance dont les contraintes et variables forment un IIS pour $k = 2$

d'affectations visitées par *procMWIC*: $a_0 = (1, 1, 2, 1, 2)$, $a_1 = (1, 2, 2, 1, 2)$, $a_2 = (1, 2, 1, 1, 2)$, $a_3 = (1, 2, 1, 2, 2)$, $a_4 = (1, 2, 1, 2, 1)$. Ces affectations violent chacune une contrainte différente de $\mathcal{C}$, respectivement c_1, c_2, c_3, c_4 et c_5. Puisque le poids de toutes ces contraintes est ensuite fixé à $|\mathcal{E}|$, celles-ci formeront un ensemble incohérent qui sera retourné par l'algorithme après seulement une itération. Par ailleurs, puisque la seule itération a produit une meilleure affectation de coût 1, on possède la garantie que K est un IIS (*voir section précédente*).

CHAPITRE 5

EXPÉRIMENTATION ET ANALYSE DES RÉSULTATS

Ce chapitre présente une série d'expériences numériques reliées aux algorithmes et autres techniques de détection décrits dans les chapitres 2 et 3. Ces expériences, qui ont été réalisées sur des instances générées aléatoirement et instances connues du problème de coloriage de graphe, ont deux buts. D'une part, ces expériences servent à analyser les avantages et désavantages de chacune de ces méthodes. D'autre part, ces expériences permettent d'évaluer les bénéfices de la détection d'IIS à l'aide de ces méthodes, particulièrement en ce qui a trait à démontrer de manière exacte l'incohérence d'un problème. Ce chapitre est organisé comme suit. Dans un premier temps, les méthodes utilisées et leur paramètres seront décrits. Ensuite, une description des instances testées sera donnée. Finalement, les expériences et leurs résultats seront présentés.

5.1 Description des méthodes et paramètres

Les algorithmes présentés au chapitre 3 font appel à deux importantes procédures: *procMWIC* et *procMHS*. La procédure *procMWIC*, utilisée par tous les algorithmes de détection, résoud le MWCSP, dans le cas de la détection d'IIS de contraintes, et le MPWCSP, dans le cas de la détection d'IIS de variables. Comme ces deux problèmes sont NP-difficiles, ces procédures ont été implantées sous la forme d'une méta-heuristique de recherche locale. Ainsi, pour l'expérimentation, l'implantation de ces procédures correspond aux méta-heuristiques de recherche Tabou, décrites dans les sections 4.1.2.1 et 4.1.2.2. Par ailleurs, il a été vu que la méta-heuristique

de recherche locale pour le MPWCSP doit, lorsque l'affectation courante a devient incohérente, désinstancier certaines variables, afin de rendre a à nouveau légale, et qu'il était, en général, difficile de déterminer le meilleur ensemble de variables à désinstancier. Une implantation gloutonne, détaillée dans l'algorithme 15 de la section 4.1.2.1, a donc été choisie pour ce problème. Par ailleurs, la procédure *procMHS*, utilisée uniquement par l'algorithme de *hitting set*, a pour but de résoudre le problème NP-difficile du *hitting set* minimum. Encore une fois, cette procédure a été implantée comme une méta-heuristique de recherche Tabou.

En choisissant une implantation heuristique pour *procMWIC*, cette procédure peut obtenir des affectations sous-optimales faisant échouer la détection d'IIS. Il a donc fallu ajouter des stratégies de récupération d'erreur aux algorithmes de détection. Afin d'éviter les échecs, l'algorithme 23 de réduction, présenté à la section 4.3.1, a été utilisé lors de l'expérimentation. La capacité de cet algorithme à éviter les erreurs peut être ajustée a l'aide des paramètres $i^{(1)}_{\max}$, $i^{(2)}_{\max}$ et τ. Les deux premiers paramètres sont le nombre maximum d'itérations allouées à *procMWIC* lors de la détection de l'ensemble incohérent K et lors de la validation de cet ensemble. Logiquement, on doit avoir une plus grande valeur pour $i^{(2)}_{\max}$ que pour $i^{(1)}_{\max}$, puisque l'étape de validation est critique à l'obtention d'un ensemble incohérent. À ces paramètres ont été ajoutés $s^{(1)}_{\max}$ et $s^{(2)}_{\max}$ qui représentent le nombre maximum de relances[1] consécutives de *procMWIC* sans amélioration de l'affectation obtenue. Puisque les chances d'obtenir une affectation optimale croîent avec le nombre de relances, on fait normalement plus de relances lors de l'étape de validation (i.e. $s^{(2)}_{\max} \geq s^{(1)}_{\max}$). Finalement, un compromis doit être fait lors du choix de ces paramètres. Ainsi, plus grand est le nombre de mouvements alloués et de relances faites, meilleures sont les chances d'avoir des affectations optimales et d'éviter les erreurs, mais plus longue est la détection. Le tableau 5.1 présente trois

[1]Chaque relance utilise une graine aléatoire différente pour *random* dans *procMWIC*.

Tableau 5.1 Différents jeux de paramètres pour *procMWIC*

Instance	$i_{\max}^{(1)}$	$i_{\max}^{(2)}$	$s_{\max}^{(1)}$	$s_{\max}^{(2)}$
facile	100000	100000	1	5
moyenne	1000000	1000000	1	5
difficile	1000000	5000000	5	10

jeux de paramètres pour *procMWIC*, que des expériences ont montré être efficaces pour des instances respectivement faciles, moyennes et difficiles à résoudre. La difficulté d'une instance particulière peut être mesurée par le nombre moyen d'itération requis pour obtenir une affectation optimale. Comme cette difficulté n'est pas connue a priori, la stratégie suivante a été adoptée. Le premier jeu de paramètres (i.e. *facile*) est d'abord utilisé pour la détection d'IIS de toutes les instances. Ensuite, pour les instances où la détection a échoué, une nouvelle tentative est faite avec le second jeu de paramètres (i.e. *moyenne*). Finalement, si la détection échoue avec ce jeu, les détections suivantes sur ces instances seront faites avec le troisième jeu de paramètres (i.e. *difficile*).

Le réglage du paramètre τ est encore plus complexe. Il a été vu que l'efficacité d'une méta-heuristique de recherche Tabou dépendait de l'ajustement du paramètre de rétention τ. Alors qu'une valeur appropriée de τ peut être évaluée pour une l'instance originale, cette valeur est susceptible de devenir inadéquate lorsque les poids des contraintes ou variables de cette instance sont modifiés par la détection. Afin d'avoir, tout au long de la détection, une valeur de τ ajustée aux poids, une stratégie d'auto-ajustement de τ a été implantée. L'algorithme 34 montre les détails de cette stratégie. Cet algorithme reçoit en paramètres l'itération courante i, le coût h de l'affectation courante, la rétention courante τ, un entier positif Δi correspondant à la période (i.e. nombre d'itérations) de l'ajustement, ainsi qu'un entier $\Delta\tau \geq 0$ et un nombre réel $g \geq 0$. À chaque itération de recherche locale,

l'algorithme met à jour le meilleur coût h_p obtenu durant la période courante. Ensuite, à la fin de chaque période (i.e. lorsque i est un multiple de Δi), l'algorithme calcule la nouvelle rétention τ' de la manière suivante. Si h_p est aussi bon que le meilleur coût h^* obtenu depuis la première itération, la rétention optimale τ^* prendra, avec une probabilité de 0.5, la valeur de τ. Par la suite, si h_p améliore le meilleur coût de la précédente période, τ' conserve la valeur de τ. Sinon, une perturbation $\Delta\tau$ est appliquée à τ. Par ailleurs, on utilise la règle suivante pour déterminer si $\Delta\tau$ doit être ajoutée ou retranchée à τ. Plus τ est éloignée de la meilleure rétention τ^*, plus il y a de chances que la perturbation rapproche ces deux valeurs. Ainsi, g détermine l'attraction de τ vers τ^*. Si g vaut 0, la perturbation est ajoutée ou retranchée selon la même probabilité. Par ailleurs, si g est infiniment grand, la perturbation rapprochant τ à τ^* est toujours choisie. Finalement, des tests ont montré que $\tau = 7$, $\Delta\tau = 1$, $g = 0.7$ sont des valeurs, pour la rétention initiale et les paramètres de cette stratégie, qui fonctionnent raisonnablement bien avec différentes instances.

L'implantation de *procMHS* consiste à fixer une taille initiale, et ensuite à chercher à l'aide d'une méta-heuristique de recherche locale un HS de cette taille. Si un HS est trouvé, la taille est réduite jusqu'à ce que la méta-heuristique soit incapable de trouver un HS de cette taille. Par contre, si la méta-heuristique est incapable de trouver un HS de la taille initiale, cette taille est augmentée jusqu'à ce qu'un HS soit finalement obtenu. Le HS retourné par *procMHS* est celui obtenu pour la plus petite taille. Par ailleurs, la stratégie Tabou consiste à interdire une variable ou une contrainte à être incluse dans le HS si celle-ci a été exclue du HS à une des dernières τ_ϵ itérations. De même, on interdit d'exclure une contrainte ou une variable qui a été incluse dans le HS, à une des τ_s dernières itérations. Il y a ainsi deux paramètres de rétention Tabou à ajuster. Au cours de l'expérimentation, ces paramètres ont été fixés à $\tau_e = 5$ et $\tau_s = 2$, après avoir constaté que ces valeurs donnaient de bons

Algorithme 34 Stratégie d'auto-ajustement de τ

Entrée: L'itération courante i;
Le coût de l'affectation courante h;
La rétention courante τ;
Un entier $\Delta i \geq 0$;
Un entier $\Delta\tau \geq 0$;
Un nombre réel $g \geq 0$;
Sortie : La nouvelle rétention τ'.

si $h < h_p$ **alors**
  $h_p \leftarrow h$;
fin si
si i est un multiple de Δi **alors**
  si $h_p \leq h^*$ et random$([0,1]) < 0.5$ **alors**
    $\tau^* \leftarrow \tau$;
  fin si
  si $h_p \geq h'_p$ **alors**
    si $\tau < \tau^*$ **alors**
      $p \leftarrow \big((\tau^* - \tau)g + 1\big)/\big((\tau^* - \tau)g + 2\big)$;
    sinon
      $p \leftarrow 1/\big((\tau - \tau^*)g + 2\big)$;
    fin si
    si random$([0,1]) < p$ **alors**
      $\tau' \leftarrow \tau + \Delta\tau$;
    sinon
      $\tau' \leftarrow \tau - \Delta\tau$;
    fin si
  sinon
    $\tau' \leftarrow \tau$;
  fin si
  $h'_p \leftarrow h_p$;
  $h_p \leftarrow h$;
fin si

résultats. Par ailleurs, le réglage des paramètres $i_{\max}$ et $s_{\max}$, qui correspondent au nombre d'itérations de recherche locale et au nombre de relances, a été fait avec comme souci la rapidité. Ainsi, comme l'algorithme de *hitting set* prend un nombre exponentiel d'itérations pour obtenir un IIS, il est important de passer le moins de temps possible à chacune de ces itérations. De plus, l'optimalité de cette procédure n'est pas aussi critique que *procMWIC*. En ce sens, si *procMHS* retourne des HS qui ne sont pas minimaux, on peut quand même obtenir un IIS, même si cet IIS n'est pas minimum. Ainsi, ces paramètres ont été fixés à $i_{\max} = 100000$ et $s_{\max} = 1$.

L'algorithme 23 de réduction requiert une implantation de la récupération d'erreur particulière à chaque méthode de détection. Ainsi, les algorithmes heuristiques de retrait, d'insertion et de *hitting set* ont été implantés suivant les descriptions données dans les sections 4.3.1.1, 4.3.1.2 et 4.3.1.3. Par ailleurs, comme les stratégies de récupération d'erreurs peuvent causer l'obtention d'un ensemble incohérent qui n'est pas un IIS, l'algorithme 32 de réductions successives a également été utilisé. On doit fournir à cet algorithme le nombre maximum $r_{\max}$ de détections (i.e. d'appels à *procRéduit*) consécutives sans réduction autorisées. Il a été constaté qu'après une deuxième détection sans réduction, l'ensemble K était, dans la plupart des cas, un IIS. Ainsi, $r_{\max} = 1$ a été employé lors de l'expérimentation.

Dans le cadre de l'expérimentation, les seuls algorithmes de détection d'IIS qui ont été retenus sont les algorithmes heuristiques de retrait, d'insertion, et de *hitting set*. Ces algorithmes ont été sélectionnés à cause de leur propriétés, de leur efficacité et de leur robustesse (i.e. récupération d'erreur). Par ailleurs, d'autres techniques ont également été retenues. Ainsi, dans certaines expériences, les algorithmes de retrait et d'insertion ont été combinés avec l'heuristique de poids du voisinage, décrite à la section 3.3.4, pour obtenir de plus petits IIS. La technique d'accélération de la section 4.4 a également été intégrée à ces deux algorithmes, dans le but

d'obtenir un IIS le plus rapidement possible. De plus, une version heuristique de l'algorithme de pré-filtrage, présenté à la section 3.3.3, a également été utilisée[2]. L'algorithme servant à obtenir une borne inférieure sur la taille d'IIS minimum, décrit à la section 3.3.5, a aussi été implanté en utilisant une méta-heuristique pour *procMWIC*. Enfin, lors de la présentation des expériences, ces algorithmes et techniques seront dénotés comme suit:

Ret: l'algorithme heuristique de retrait;

Ret+h: *Ret* combiné avec l'heuristique de poids de voisinage;

Ins: l'algorithme heuristique d'insertion;

Ins+h: *Ins* combiné avec l'heuristique de poids de voisinage;

HS: l'algorithme heuristique de *hitting set*;

FI: l'algorithme de pré-filtrage suivi de *Ins+h*;

LB: l'algorithme de borne inférieure de la section 3.3.5;

Le but de certaines expériences est d'évaluer l'utilité de la détection d'IIS pour résoudre des problèmes complexes. Ainsi, des IIS sont obtenus pour des instances du problème de k-coloriage de graphe, dans le but de déterminer le nombre chromatique de ces graphes. Ces expériences nécessitent d'avoir une méthode exacte qui, étant donné un graphe, retourne le nombre chromatique de ce graphe. La méthode exacte utilisée pour ces expériences est un algorithme de retour-arrière décrit dans (Peemöller, 1983). Cet algorithme est une version corrigée de la modification apportée par Brélaz à l'algorithme de Brown (Brélaz, 1979; Brown, 1972).

[2]L'implantation de la version heuristique est similaire à celle de l'algorithme d'insertion.

Finalement, il est important de mentionner que toutes les expériences numériques, présentées dans ce chapitre, ont été réalisées sur des ordinateurs ayant un processeur AMD Athlon de 1.6Ghz et 512Mo de mémoire vive.

5.2 Description des données

Les expériences, portant sur le problème de coloriage de graphes, ont été réalisées sur deux jeux d'instances. Le premier jeu contient des instances générées aléatoirement. Étant donné un entier positif n et un nombre réel $p \in [0, 1]$, un graphe (n, p) aléatoire est tel que $|V| = n$ et chacune des $n(n-1)/2$ paires de sommets a une probabilité p d'être reliée par une arête de E. Le paramètre p est donc la densité espérée d'arête du graphe. Comme convention, on dénote "R⟨n⟩.⟨p⟩" un graphe (n, p) aléatoire particulier, généré dans l'expérimentation.

Le second jeu d'instances contient les graphes de références du deuxième challenge *DIMACS* portant sur le coloriage de graphe (Johnson et Trick, 1996). Ces graphes, qui proviennent de sources variées, ont des propriétés particulières. Voici une brève description des différents types d'instances de ce jeu; pour plus d'informations, se référer à `http://mat.gsia.cmu.edu/COLOR04`:

DSJ: Graphes aléatoires utilisés par Aragon et al. dans (Aragon et al., 1991);

- **DSJC:** Graphes (n, p) aléatoires;
- **DSJR:** Graphes géométriques;
- **DSJRc:** Compléments de graphes géométriques;

REG: Graphes basés sur l'allocation de registres pour des variables dans le code de programmes réels;

LEI: Graphes de Leighton pour lesquels le nombre chromatique équivaut à la taille de la plus grande clique;

SCH: Graphes d'horaires de cours:

SGB: Graphes provenant de (Knuth, 1993). Ces graphes peuvent être divisés dans les catégories suivantes:

Book: Étant donné une oeuvre littéraire, le graphe correspondant possède un sommet par personnage, et une arête pour chaque paire de personnages qui se sont rencontrés dans l'oeuvre;

Game: Graphes représentant les parties disputées dans une saison de football collégial. Ces graphes possèdent un sommet par équipe de la ligue, et une arête pour chaque paire d'équipes qui se sont affrontées durant la saison;

Miles: Graphes représentant une carte routière des États-Unis. Ces graphes possèdent un sommet par ville importante, et une arête pour chaque paire de villes suffisamment rapprochées;

Queen: Étant donné un échiquier $m \times n$, le graphe correspondant possède un sommet par case de l'échiquier et une arête pour chaque paire de cases situées sur la même rangée, colonne ou diagonale de l'échiquier. Si $m = n$, on peut démontrer que le nombre chromatique du graphe est toujours égal à n, dans les cas ou n est impair où multiple de 3 (e.g. $n = 1, 3, 5, 6, 7, 9, 11, \ldots$);

MYC : Graphes basés sur la transformation de Mycielski. Ces graphes sont difficiles à résoudre car ils ne contiennent pas de triangles, mais leur nombre chromatique augmente avec leur taille;

MIZ : Graphes possédant une clique de quatre sommets difficile à trouver;

HOS : Graphes obtenus d'un problème de partitionnement de matrices;

CAR : Graphes généralisant les graphes *MYC* avec l'ajout de sommets pour accroître leur taille sans changer leur densité;

5.3 Détection d'IIS de contraintes versus variables

La première expérience a pour but de comparer la détection d'IIS de contraintes et de variables sur un graphe $(50, 0.5)$ aléatoire, R50.5, possédant 590 arêtes de nombre chromatique 9. Le tableau 5.2 montre les résultats de la détection sur R50.5 en utilisant trois algorithmes: *Ret+h*, *Ins+h* et *FI*. Pour chacun de ces algorithmes, 10 IIS de variables et 10 de contraintes ont été obtenus pour le problème de 8-coloriage de R50.5, utilisant différentes graines aléatoires pour la procédure *random*. Le nombre moyen de variables et de contraintes des CN résultant de ces IIS est respectivement donné dans les colonnes $|\mathcal{X}'|$ et $|\mathcal{C}'|$, et la densité de contraintes résultante dans la colonne p'. Comme pour la densité d'arête d'un graphe, p' a été calculée comme suit:

$$p' = \frac{2\ |\mathcal{C}'|}{|\mathcal{X}'|\ (|\mathcal{X}'| - 1)}$$

Le nombre chromatique des graphes correspondant à ces IIS a ensuite été obtenu avec la méthode exacte (*voir section 5.1*), après un nombre moyen de retour-arrières inscrit dans la colonne RA'. Ces résultats montrent que la détection fonctionne différemment dans le cas d'IIS de variables que pour des IIS de contraintes. Par exemple, *FI* produit, en moyenne, les plus petits IIS de variables des trois algorithmes, mais également les plus gros IIS de contraintes. À l'opposé, *Ret+h* donne, en moyenne, les plus gros IIS de variables, alors que les plus petits IIS de contraintes ont aussi été obtenus par ce même algorithme. Des différences apparaissent également entre les IIS de variables et de contraintes trouvés par les

algorithmes de détection. Alors que les CN provenant des IIS de contraintes ont moins de contraintes que ceux des IIS de variables, les CN provenant des IIS de variables ont moins de variables. En conséquence, la densité de contraintes des CN d'IIS de contraintes est bien inférieure à celle des CN d'IIS de variables (0.44 en moyenne pour les IIS de contraintes comparé à 0.54 pour les IIS de variables). On remarque également que la densité de contraintes des CN d'IIS de variables est supérieure à la densité $p = 0.5$ de R50.5. Cette augmentation de la densité est probablement due à l'heuristique de poids de voisinage et à l'algorithme de pré-filtrage qui trouvent des petits IIS dans des régions plus denses de l'instance. Par ailleurs, un résultat moins prévisible est l'énorme différence dans le nombre de retour-arrières pour les IIS de variables et de contraintes (367 en moyenne pour les IIS de variables comparé à 1670 pour les IIS de contraintes). Cet écart s'explique, en majeure partie, par la différence dans la densité de contraintes des CN correspondants. Tout d'abord, les CN d'IIS de variables ont moins de variables, donnant un plus petit espace de recherche. Ensuite, le plus grand nombre de contraintes dans ces CN permet d'éliminer plus de solutions de l'espace de recherche, réduisant ainsi le nombre de retour-arrières.

Lorsque le but est de trouver une borne inférieure au nombre chromatique d'un graphe, on constate qu'il est plus facile de le faire en cherchant des IIS de variables, que des IIS de contraintes. En plus de produire des IIS dont les graphes correspondants sont plus faciles à résoudre par la méthode exacte, la détection prend moins de temps pour des IIS de variables que des IIS de contraintes. Ainsi, dans le cas de l'algorithme de retrait, le nombre d'itérations pour obtenir un IIS est, dans le pire cas, égal à $|V|$ pour un IIS de variable et à $|E|$ pour un IIS de contraintes. De plus, dans le cas de l'algorithme d'insertion, le nombre d'itérations vaut, dans le pire cas, $|\mathcal{X}'|$ pour un IIS de variables et $|\mathcal{C}'|$ pour un IIS de contraintes. Par exemple, la détection d'IIS de variables avec *Ins+h* a nécessité, en

moyenne, 37 itérations, alors que 345 itérations ont été requises pour la détection d'IIS de contraintes avec ce même algorithme. Pour ces raisons, les expériences suivantes se concentrent exclusivement sur la détection d'IIS de variables.

Tableau 5.2 Détection d'IIS de variables et de contraintes sur l'instance *R50.5*

	IIS de variables				IIS de contraintes			
Détection	$\lvert\mathcal{X}'\rvert$	$\lvert\mathcal{C}'\rvert$	p'	RA$'$	$\lvert\mathcal{X}'\rvert$	$\lvert\mathcal{C}'\rvert$	p'	RA$'$
Ret+h	38.2	384.1	0.54	380.4	36.8	327.5	0.50	1089.4
Ins+h	37.1	355.7	0.53	348.5	41.2	344.8	0.42	1295.4
FI	36.3	344.7	0.54	371.8	41.9	344.4	0.40	2625.4

5.4 Détection d'IIS avec l'algorithme de *hitting set*

La prochaine expérience évalue l'algorithme de *hitting set*. Pour cette expérience des graphes $(n, 0.5)$ aléatoires ont été générés pour chaque $n \in \{15, 20, 25, 30, 35, 40, 45, 50\}$. L'algorithme de *hitting set* a ensuite été appliqué à la détection de 10 IIS de variables pour le problème de $(k-1)$-coloriage de chacun de ces graphes, utilisant à chaque fois une graine aléatoire différente pour *random*. Le tableau 5.3 donne les résultats de cette expérience. Les trois premières colonnes fournissent le nombre de variables $|\mathcal{X}|$ et de contraintes $|\mathcal{C}|$ de l'instance originale, ainsi que la valeur de k utilisée pour le problème. Cette valeur correspond au nombre chromatique du graphe, obtenu avec la méthode exacte. Enfin, les deux dernières colonnes du tableau donnent le nombre moyen de variables $|\mathcal{X}'|$ des IIS obtenus et le nombre moyen d'itérations mis par l'algorithme de *hitting set* pour obtenir ces IIS.

Tableau 5.3 L'algorithme de *hitting set* sur des instances aléatoires de densité 0.5.

Instances			IIS de var.	
$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	k	$\|\mathcal{X}'\|$	*iter.*
15	46	4	4	10.9
20	90	5	5	16.6
25	137	6	9	40.4
30	211	7	7	22.2
35	277	7	7	41.7
40	369	8	25	641.5
45	473	9	42	164.8
50	590	9	32	2554.4

On remarque, tout d'abord, que l'algorithme de *hitting set* a trouvé, pour chaque instance, 10 IIS ayant le même nombre de variables, ce qui indique que ces IIS sont fort probablement minimum. Par ailleurs, les résultats montrent également que le nombre d'itérations de détection est, comme prévu, exponentiel par rapport à $|\mathcal{X}|$. On constate, cependant, que cette relation n'est pas strictement croissante, puisque la valeur *iter* baisse temporairement lorsque k augmente. Ce phénomène sera expliqué en détails lors d'une expérience subséquente (*voir section 5.7*).

5.5 Influence des heuristiques de détection

La prochaine expérience analyse l'influence des heuristiques sur la détection d'IIS de variables sur un ensemble varié d'instances. Le tableau 5.4 présente les résultats de cette expérience. Les cinq premières colonnes donnent le nom, le type, le nombre de variables et le nombre de contraintes de l'instance, ainsi que la valeur de k utilisée pour le problème. Cette valeur correspond au nombre chromatique du graphe, obtenu avec la méthode exacte. Les colonnes suivantes contiennent le nombre minimum, médian et maximum de variables de 10 IIS trouvés[3] par les algorithmes

[3]Une graine aléatoire différente a été utilisée pour chaque IIS.

Ret, *Ret+h*, *Ins*, *Ins+h*, et *FI*, pour le problème de $(k-1)$-coloriage de l'instance correspondante.

Tableau 5.4 Détection d'IIS de variables avec et sans heuristique

Instances					IIS de variables				
					Ret	Ret+h	Ins	Ins+h	FI
nom	type	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	k	$\|\mathcal{X}'\|$	$\|\mathcal{X}'\|$	$\|\mathcal{X}'\|$	$\|\mathcal{X}'\|$	$\|\mathcal{X}'\|$
R50.5	(n,p)	50	590	9	36,41,44	36,38,41	37,41,44	34,37,39	32,36,40
R60.5	(n,p)	60	858	10	48,51,53	44,46,48	49,52,54	45,46,48	43,44,46
DSJC125.1	DSJC	125	736	5	10,31,50	10,10,13	68,84,87	10,14,53	11,14,35
queen6_6	Queen	36	290	7	25,27,29	26,27,27	27,28,30	24,27,28	22,24,27
queen8_8	Queen	64	728	9	57,58,59	54,55,56	56,58,60	54,55,56	53,55,56
queen9_9	Queen	81	2112	10	-	73,74,74	-	73,74,75	73,75,76

Ces résultats indiquent clairement que les algorithmes de retrait et d'insertion donnent de plus petits IIS lorsqu'ils sont combinés avec l'heuristique de poids de voisinage. Ainsi, dans tous les cas sauf un (i.e *Ret* sur *queen6_6*), les plus petits IIS trouvés utilisant l'heuristique ont un nombre inférieur ou égal de variables à ceux obtenus sans heuristique. De plus, le nombre médian de variables des IIS obtenus avec cette heuristique est strictement inférieur pour toutes les instances sauf une (i.e. encore *Ret* sur *queen6_6*), tandis que les plus gros IIS trouvés avec l'heuristique ont moins de variables pour toutes les instances. L'heuristique de poids de voisinage permet donc de réduire la variance dans la taille des IIS obtenus. Un bon exemple est l'instance *DSJC125.1* qui contient des IIS minimum de seulement 10 variables. Pour cette instance l'algorithme de retrait sans heuristique a trouvé un tel IIS minimum dans 2 des 10 cas, alors que le même algorithme avec l'heuristique en a trouvé un dans 6 cas. De même, le plus gros IIS obtenu par *Ret+h* a 13 variables, comparé à 50 pour *Ret*.

On constate, par ailleurs, que l'algorithme de pré-filtrage performe encore mieux comme heuristique pour trouver de petits IIS. Ainsi, pour les instances *R50.5*, *R60.5*, *queen6_6* and *queen8_8*, *FI* a trouvé des IIS contenant moins de variables que ceux obtenus par n'importe quel autre algorithme de détection. De plus, pour les instances *R50.5* et *queen6_6*, ces IIS ont été montrés minimum par l'algorithme de *hitting set* (*voir l'expérience suivante*). En somme, considérant que l'algorithme de pré-filtrage permet également d'accélérer la détection d'un IIS, cet algorithme est probablement le meilleur choix pour obtenir de petits IIS.

5.6 Bornes inférieures sur la taille d'IIS minimum

Cette expérience a pour but de comparer les bornes inférieures sur la taille d'IIS minimum obtenues par l'algorithme de *hitting set* et celui présenté à la section 3.3.5. Les résultats de cette expérience sont présentés dans le tableau 5.5. Les cinq premières colonnes donnent le nom, le type, le nombre de variables et le nombre de contraintes de l'instance, ainsi que la valeur de k utilisée pour le problème, correspondant au nombre chromatique du graphe, obtenu avec la méthode exacte. Les deux dernières colonnes donnent le nombre minimum, médian et maximum de variables de 10 IIS trouvés par *HS* pour le problème de $(k-1)$-coloriage de l'instance correspondante, et de 10 bornes inférieures obtenues par *LB*. Dans le cas de *HS*, les valeurs précédées de "$\geq$" représentent des bornes inférieures obtenues en arrêtant l'algorithme après 3000 itérations sans obtenir d'IIS[4]. Ces résultats montrent que *HS* performe mieux que *LB* pour obtenir une borne inférieure. Ainsi, pour toutes les instances testées avec *HS*, les bornes obtenues sont plus élevées que celles trouvées pas *LB*. Par exemple, *HS* a trouvé une borne de 36 variables pour l'instance *R60.5*, alors que la meilleure borne obtenue par *LB* est de seulement 11

[4]Les bornes correspondent à la taille du dernier HS.

variables. De plus, *HS* a trouvé des bornes optimales pour les instances *R50.5* and *queen6_6*. Enfin, lorsque la taille de l'instance permet son utilisation, *HS* produit de meilleures bornes inférieures que *LB*.

Tableau 5.5 Bornes sur la taille d'IIS minimum de variables

Instances					IIS de variables	
					HS	LB
nom	type	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	k	$\|\mathcal{X}'\|$	$\|\mathcal{X}'\|$
R50.5	(n,p)	50	590	9	32,32,32	13,13,13
R60.5	(n,p)	60	858	10	≥ 36	10,10,11
DSJC125.1	DSJC	125	736	5	≥ 10	4,4,5
queen6_6	Queen	36	290	7	22,22,22	7,7,7
queen8_8	Queen	64	728	9	≥ 29	11,11,12
queen9_9	Queen	81	2112	10	-	12,12,13

5.7 Détection d'IIS sur les instances aléatoires

La prochaine expérience cherche à démontrer l'utilité de la détection d'IIS de variables pour obtenir de manière exacte une borne inférieure sur le nombre chromatique de graphes aléatoires. L'approche utilisée est similaire à celle proposée par Herrmann et Hertz (*voir section 2.4*). Soit un graphe (n,p) aléatoire G. La méthode exacte est d'abord appliquée sur ce graphe afin de déterminer son nombre chromatique $\chi(G)$. Dans les cas où la méthode exacte n'a pas trouvé $\chi(G)$ dans un temps alloué raisonnable, une borne supérieure sur $\chi(G)$ est obtenue à l'aide d'une méta-heuristique. Par exemple, on peut convertir le problème de k-coloriage du graphe en CN, et déterminer la plus petite valeur de k telle qu'une des méta-heuristiques présentées à la section 4.1 obtienne une affectation de coût nul. Ensuite, un algorithme de détection est appliqué au problème de $(k-1)$-coloriage pour obtenir un IIS de variables. Finalement, la méthode exacte est appliquée au

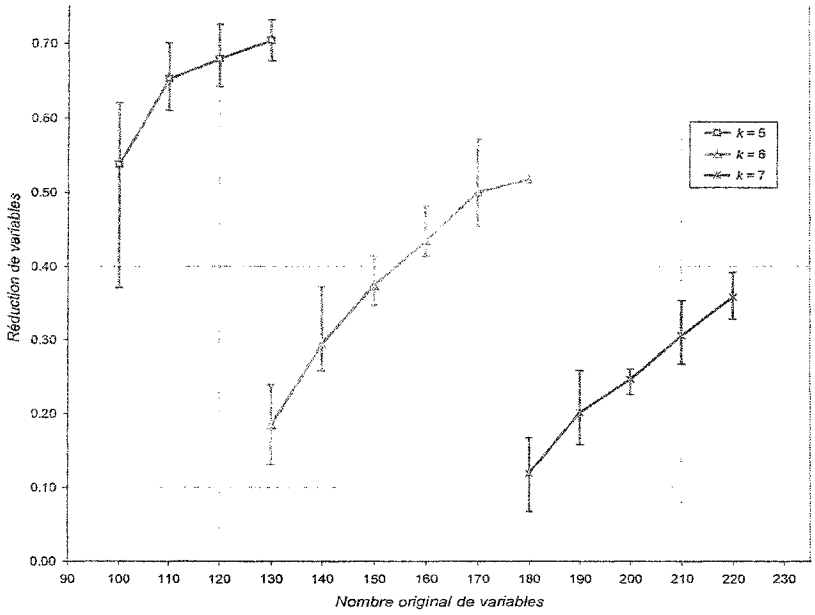

Figure 5.1 Réduction de variables pour des instances $(n, 0.1)$ aléatoires versus n

graphe G' induit par les sommets correspondant aux variables de l'IIS. Comme ce graphe possède possiblement moins de sommets et d'arêtes que G, le nombre de retour-arrières nécessaires à la méthode exacte pour déterminer $\chi(G')$ devrait également être réduit. Si la méthode exacte parvient à déterminer $\chi(G')$, cette valeur correspond alors à $\chi(G)$. Sinon, le même processus est répété pour une plus petite valeur de k, qui est alors une borne inférieure à $\chi(G)$. Par ailleurs, puisqu'ils ne possèdent aucune structure particulière, les graphes (n, p) aléatoires testés dans cette expérience sont des instances qui se portent mal à la détection d'IIS. Ainsi, pour des valeurs de p plus élevées, de tels graphes peuvent contenir des IIS ayant presque le même nombre de variables.

Les tableaux I.1 et I.2 montrent les résultats de la détection d'IIS de variables sur des graphes aléatoires pour p valant 0.1 et 0.5. Ainsi, quatre graphes aléatoires ont été générés pour chaque paire $(n, 0.1)$ avec $n \in \{100, 110, \ldots, 220\}$, et pour chaque

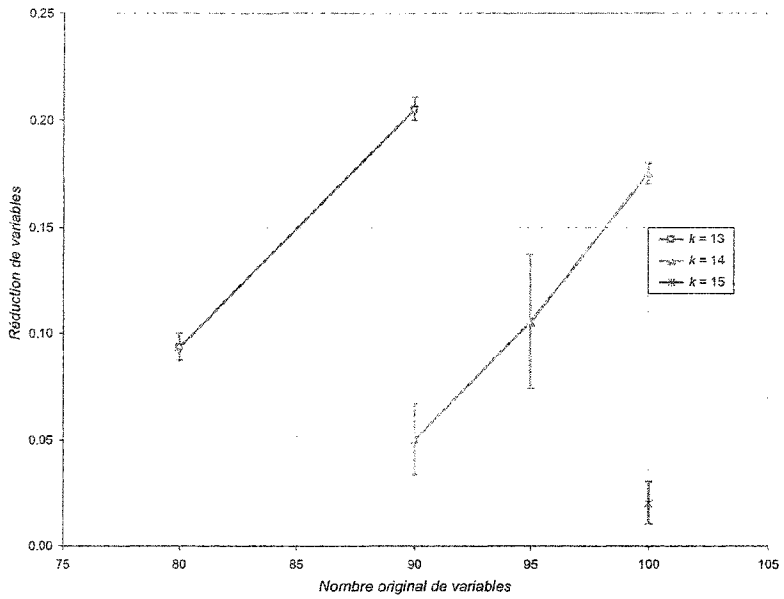

Figure 5.2 Réduction de variables pour des instances $(n, 0.5)$ aléatoires versus n

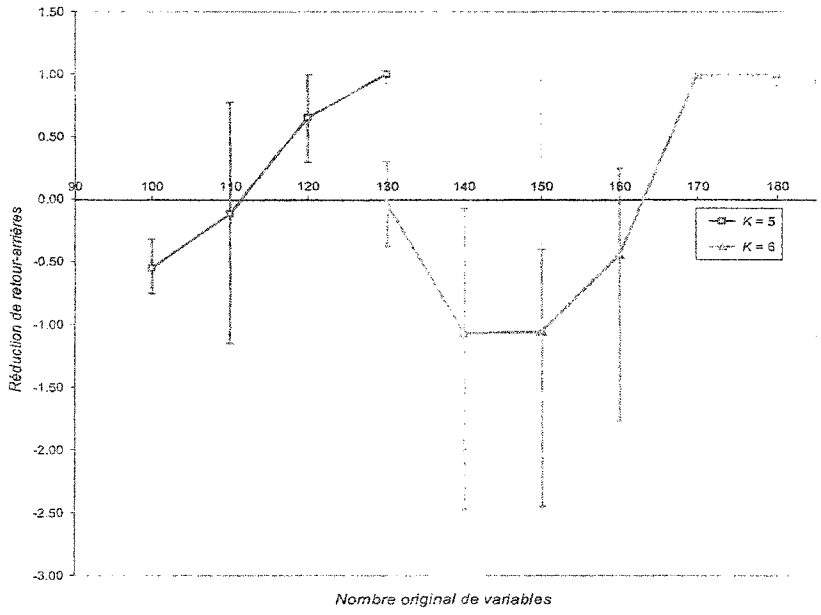

Figure 5.3 Réduction de retour-arrières pour des instances $(n, 0.1)$ aléatoires versus n

paire $(n, 0.5)$ avec $n \in \{80, 90, 95, 100\}$. Les trois premières colonnes de ces tableaux contiennent le nombre de variables et de contraintes, ainsi que la borne $k \geq \chi(G)$ utilisée pour le problème de $(k-1)$-coloriage de ces graphes. La quatrième colonne donne le nombre de retour-arrières requis à la méthode exacte pour déterminer $\chi(G)$. Les valeurs données sans parenthèses indiquent que $\chi(G)$ a pu être calculé sans dépasser la limite des 250 000 000 retour-arrières permis. Dans ces cas, $k = \chi(G)$. Par contre, les valeurs comprises entre parenthèses représentent le temps de calcul (en secondes) qu'il a fallu à la méthode exacte pour atteindre la limite de retour-arrières. Dans de tels cas, k est une borne supérieure de $\chi(G)$, obtenue à l'aide d'une méta-heuristique. Par ailleurs, les autres colonnes contiennent le nombre de variables et de contraintes du CN provenant du plus petit IIS de 10 IIS obtenus avec *Ret+h*, *Ins+h* et *FI*, ainsi que le nombre de retour-arrières requis pour déterminer le nombre chromatique des graphes correspondants. Encore une fois, les valeurs comprises entre parenthèses représentent les temps de calcul de la méthode exacte pour dépasser la limite de retour-arrières. Le nombre chromatique de ces graphes réduits peut alors être strictement inférieur à k.

Les figures 5.1 et 5.2 ont été produites avec les IIS de variables obtenus par *FI*[5]. Chaque courbe contient des instances pour une valeur particulière de k, et est tracée de telle sorte que les valeurs en abscisse sont le nombre de variables $|\mathcal{X}|$ de ces instances, et les valeurs en ordonnée sont les réductions minimum, moyenne et maximum des variables pour les IIS correspondants. Étant donné un IIS de $|\mathcal{X}'|$ variables, la réduction de variables est calculée comme suit:

$$\frac{|\mathcal{X}| - |\mathcal{X}'|}{|\mathcal{X}|}$$

Ces figures montrent, tout d'abord que la réduction de variables diminue lorsque p

[5]Les IIS trouvés par *Ret+h* et *Ins+h* donnent des courbes similaires.

augmente. Ainsi, la réduction maximum atteinte pour les instances de 100 variables est de 62% lorsque $p = 0.1$, alors que la réduction maximum pour $p = 0.5$ est seulement de 18%. Ce résultat n'est pas surprenant, puisque les instances ayant une plus grande densité de contraintes ont généralement de plus gros IIS. De plus, ces figures révèlent deux tendances contradictoires, lorsqu'on les considère séparément. D'un côté, la réduction de variables décroît lorsque k augmente. Considérons, par exemple, les valeur de réduction pour les courbes de la figure 5.1. Pour $k = 5$, la réduction maximum de variables est de 73%, tandis que cette valeur baisse à 57% pour $k = 6$, et 39% pour $k = 7$. De même, pour les courbes de la figure 5.2, la réduction maximum de variables est de 21% pour $k = 13$, 18% pour $k = 14$, et 3% pour $k = 15$. D'un autre côté, la réduction de variables augmente avec n, pour une valeur particulière de k. Considérons à nouveau la figure 5.1. Pour $k = 5$, la réduction maximum de variables passe de 62%, lorsque $n = 100$, a 73%, pour $n = 130$. De la même manière, pour $k = 6$ la réduction passe de 24%, pour $n = 130$ à 57% pour $n = 170$. La même chose se produit pour $k = 7$, où la réduction passe de 7% à 38%, lorsque n varie de 180 à 220.

La figure 5.3 montre la réduction minimum, moyenne et maximum de retour-arrières de la méthode exacte pour les IIS obtenus avec une valeur particulière de k. Pour $k = 5$, la réduction maximum de retour-arrières est de -33% lorsque $n = 100$ (i.e. le nombre de retour-arrières est supérieur pour le graphe réduit). Cependant, la réduction maximum augmente à une valeur raisonnable de 77% pour $n = 110$, et presque 100% pour $n = 120$ et $n = 130$. Pour $k = 6$, la réduction maximum de retour-arrières commence à une valeur positive de 30% pour $n = 130$, mais tombe à -8% pour $n = 140$ et jusqu'à -40% pour $n = 150$. Heureusement, la réduction maximum remonte ensuite à une valeur positive de 24% pour $n = 160$ et atteint près de 100% pour $n = 170$ et $n = 180$. Ces résultats suggèrent que la détection d'IIS est particulièrement utile pour déterminer le nombre chromatique d'instances

ayant le plus possible de variables pour une valeur de k donnée.

Dans la plupart des problèmes combinatoires, il existe une brusque transition dans la taille des instances qui peuvent être résolues optimalement et celles qui ne peuvent pas l'être. Dans le cas de graphes $(n, 0.1)$ aléatoires, la méthode exacte utilisée dans l'expérimentation résoud toutes les instances de 160 variables en moins de 200 000 retour-arrières, alors que deux des quatre instances de 170 variables n'ont pas été résolues après 250 000 000 retour-arrières, et aucune des instances de 180 variables n'ont été résolues dans cette même limite. Cependant, comme ces instances possèdent un nombre de variables situé à la limite supérieure pour $p = 0.1$ et $k = 6$ (i.e. toutes les instances de plus de 180 variables ont $k = 7$), elles sont d'excellents candidats pour la détection d'IIS. Ainsi, les deux instances de 170 variables qui n'ont été résolues dans la limite de retour-arrières ont produit des IIS donnant des graphes facilement résolus en 167 544 et 7367 retour-arrières. De même, pour l'instance de 180 variables pour $k = 6$, l'IIS obtenu a donné un graphe qui a été résolu en seulement 10 347 retour-arrières.

On constate, finalement, que les résultats de cette expérience révèlent un phénomène surprenant. En réduisant le nombre de variables d'une instance donnée, on pourrait s'attendre à ce que la méthode exacte, dont la complexité est exponentielle au nombre de variables, résoud cette instance en un nombre inférieur ou égal de retour-arrières. Cependant, la figure 5.3 montre clairement que cela n'est pas toujours le cas. Par exemple, pour les instances de 150 variables, tous les IIS trouvés ont donné des graphes pour lesquels le nombre de retour-arrières a augmenté, au lieu de diminuer. Un exemple frappant est celui d'un IIS de 103 variables obtenu par *Ins+h* pour une instance de 160 variables (i.e. réduction de variables de 36%). Alors que la méthode exacte a mis 199 720 retour-arrières pour résoudre l'instance originale, 7 955 344 retour-arrières ont été nécessaires pour l'instance réduite (i.e. augmentation de 3883% des retour-arrières). Ce phénomène, déjà observé par

Herrmann et Hertz dans (Herrmann et Hertz, 2002), pourrait s'expliquer par le comportement de l'algorithme exact de coloriage utilisé. Ainsi, cet algorithme utilise une heuristique qui colore d'abord le sommet ayant la plus grande valeur de saturation (i.e. le nombre de couleurs différentes utilisées sur les sommets voisins). Cette heuristique recherche, indirectement, un petit ensemble de sommets pour lesquels aucune coloration n'existe. En trouvant un IIS dans une instance, les algorithmes de détection se trouvent, en même temps, à éliminer tous les autres IIS de l'instance. À l'exception de l'algorithme de *hitting set* qui trouve des IIS minimum, les algorithmes de détection éliminent donc de plus petits IIS, augmentant ainsi le nombre de retour-arrières.

5.8 Détection d'IIS sur les instances de référence

La dernière expérience consiste à obtenir des IIS afin de déterminer une borne inférieure sur le nombre chromatique d'instances de référence du challenge DIMACS. Les tableaux 5.6 5.7 et 5.8 présentent les résultats de cette expérience. Les cinq premières colonnes de ces tableaux contiennent le nom, le type, le nombre de variables et de contraintes, ainsi que le nombre de retour-arrières requis par la méthode exacte pour déterminer $\chi(G)$. Les valeurs comprises entre parenthèses représentent le nombre de retour-arrières exécutés par la méthode exacte avant d'atteindre la limite de 4 heures de temps de calcul. Dans ces cas, la méthode exacte n'a pas pu déterminer $\chi(G)$. La colonne suivante donne une borne inférieure $k \leq \chi(G)$. Les valeurs de k précédées d'un astérisque "$*$" indiquent que $\chi(G)$ est connu, et que $k = \chi(G)$. Les deux colonnes suivantes contiennent le nombre de variables et de contraintes de CN provenant du plus petit IIS de variables obtenu en 5 essais[6] pour le problème de $(k-1)$-coloriage de l'instance correspondante. Ensuite,

[6]Utilisant chaque fois une graine aléatoire différente pour *random*.

l'avant-dernière colonne donne le nombre de retour-arrières mis par la méthode exacte pour déterminer le nombre chromatique de l'instance réduite. Une fois de plus, les valeurs comprises entre parenthèses correspondent aux cas où ce nombre chromatique n'a pu être obtenu à l'intérieur de la limite de 4 heures de temps de calcul, et pour lesquels ce nombre chromatique peut être strictement inférieur à k. Enfin, la dernière colonne contient des bornes inférieures sur la taille d'IIS minimum des instances testées, obtenues par LB.

Afin de faciliter la présentation les résultats de cette expérience, les instances seront divisées en trois catégories. La première catégorie est composée d'instances dont l'ensemble de variables $\mathcal{X}$ est probablement un IIS au problème de $(k-1)$-coloriage de graphe (i.e. instances provenant de graphes sommet-critiques). Les instances de type *MYC*, *MIZ*, ainsi que celles dont le nom contient *Insertions* sont dans cette catégorie. Les résultats du tableau 5.6 montrent que pour toutes les instances de cette catégorie, un IIS $\mathcal{X}' = \mathcal{X}$ a été obtenu. De plus, pour les 12 instances où $\chi(G)$ est connu, les expériences ont permis d'observer que ces instances proviennent de graphes sommet-critiques. Pour les 6 autres instances, il a été démontré que $k < \chi(G)$ ou que ces instances proviennent également de graphes sommet-critiques. Par ailleurs, comme la technique d'accélération de la section 4.4 est utilisée en combinaison avec l'algorithme de détection, les IIS ont été obtenus rapidement, même pour les instances ayant un grand nombre de variables. Ainsi, dans le cas de l'instance *3-Insertions_5* possédant 1406 variables, l'IIS a été obtenu en seulement 22 itérations[7]. La technique d'accélération est donc particulièrement utile pour montrer que l'ensemble des variables ou des contraintes d'une instance est un IIS. Quand aux bornes inférieures obtenues par LB, ces bornes sont généralement près de $|\mathcal{X}|$. Par exemple, une borne optimale valant $|\mathcal{X}|$ a été obtenue pour toutes les instances de type *MYC* sauf une.

[7] 1152 variables de l'IIS ont été trouvées après la première itération, et 1372 après la seconde.

Tableau 5.6 Détection d'IIS de variables sur la première catégorie d'instances du challenge DIMACS

Instances						IIS de variables			
nom	type	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	RA	k	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	LB
myciel3	MYC	11	20	4	*4	11	20	-	11
myciel4	MYC	23	71	106	*5	23	71	-	23
myciel5	MYC	47	236	30998	*6	47	236	-	47
myciel6	MYC	95	755	(138446852)	*7	95	755	-	95
myciel7	MYC	191	2360	(77223695)	*8	191	2630	-	189
mug88_1	MIZ	88	146	2204467	*4	88	146	-	55
mug88_25	MIZ	88	146	942961	*4	88	146	-	56
mug100_1	MIZ	100	166	1406570	*4	100	166	-	67
mug100_25	MIZ	100	166	974170	*4	100	166	-	68
1-Insertions_4	CAR	67	232	104296036	*5	67	232	-	67
1-Insertions_5	CAR	202	1227	(133727661)	6	202	1227	-	202
1-Insertions_6	CAR	607	6337	(50929137)	7	607	6337	-	448
2-Insertions_3	CAR	37	72	3064	*4	37	72	-	37
2-Insertions_5	CAR	597	3936	(48458541)	6	597	3936	-	208
3-Insertions_3	CAR	56	110	723616	*4	56	110	-	56
3-Insertions_4	CAR	281	1046	(95076991)	5	281	1046	-	220
3-Insertions_5	CAR	1406	9695	(13784327)	6	1406	9695	-	73
4-Insertions_4	CAR	475	1795	(70891706)	5	475	1795	-	232

La seconde catégorie renferme les instances pour lesquelles les IIS minimum sont les sommets de cliques pour $k = \chi(G)$. Les instances de type *Book*, *REG*, *LEI*, *SCH*, *Game* et *Miles* sont toutes dans cette catégorie. Le tableau 5.7 présente les résultats pour ces instances. Ces instances sont de bons candidats à la détection d'IIS parce qu'elles ont de très petits IIS (i.e. sommets de cliques) qui sont généralement faciles à détecter. De plus, comme le nombre chromatique d'une clique est égal à son nombre de sommets, la méthode exacte n'est pas du tout nécessaire. On remarque que ces IIS minimaux ont été trouvés dans chacune des 39 instances de cette catégorie, parmi lesquelles 17 n'avaient pas été résolues par la méthode exacte. Une fois de plus, *LB* donne des bornes généralement voisines de la taille des IIS minimum.

Tableau 5.7 Détection d'IIS de variables la deuxième catégorie d'instances du challenge DIMACS

Instances						IIS de variables			
nom	type	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	RA	k	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	LB
fpsol2.i.1	REG	496	11654	(169107715)	*65	65	2080	0	24
fpsol2.i.2	REG	451	8691	2	*30	30	435	0	24
fpsol2.i.3	REG	425	8688	2	*30	30	435	0	24
inithx.i.1	REG	864	18707	1	*54	54	1431	0	41
inithx.i.2	REG	645	13979	(139157853)	*31	31	465	0	25
inithx.i.3	REG	621	13969	(141407783)	*31	31	465	0	25
mulsol.i.1	REG	197	3925	1	*49	49	1176	0	44
mulsol.i.2	REG	188	3885	6	*31	31	465	0	24
mulsol.i.3	REG	184	3916	6	*31	31	465	0	25
mulsol.i.4	REG	185	3946	(161605284)	*31	31	465	0	24
mulsol.i.5	REG	186	3973	(161214648)	*31	31	465	0	28
zeroin.i.1	REG	211	4100	24	*49	49	1176	0	44
zeroin.i.2	REG	211	3541	11472	*30	30	435	0	27
zeroin.i.3	REG	206	3540	11472	*30	30	435	0	27
le450_15a	LEI	450	8168	(54447597)	*15	15	105	0	7
le450_15b	LEI	450	8169	(49996287)	*15	15	105	0	7
le450_15c	LEI	450	16680	(40481025)	*15	15	105	0	3
le450_15d	LEI	450	16750	(35180270)	*15	15	105	0	3
le450_25a	LEI	450	8260	14	*25	25	300	0	20
le450_25b	LEI	450	8263	12	*25	25	300	0	19
le450_25c	LEI	450	17343	(41188964)	*25	25	300	0	7
le450_25d	LEI	450	17425	(42974825)	*25	25	300	0	7
le450_5a	LEI	450	5714	(21467721)	*5	5	10	0	2
le450_5b	LEI	450	5734	(28479480)	*5	5	10	0	2
le450_5c	LEI	450	9803	5	*5	5	10	0	2
le450_5d	LEI	450	9757	5754158	*5	5	10	0	2
school1	SCH	385	19095	17	*14	14	91	0	2
school1_nsh	SCH	352	14612	(59393984)	14	14	91	0	2
anna	Book	138	493	8	*11	11	55	0	11
david	Book	87	406	36	*11	11	55	0	11
homer	Book	561	1629	(244497375)	*13	13	78	0	13
huck	Book	74	301	211680	*11	11	55	0	11
jean	Book	80	254	8645	*10	10	45	0	10
games120	Game	120	638	(516246020)	*9	9	36	0	9
miles1000	Miles	128	3216	4583894	*42	42	861	0	41

... continué sur la page suivante

Instances						IIS de variables			
nom	type	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	RA	k	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	LB
miles1500	Miles	128	5198	1692256	*73	73	2628	0	73
miles250	Miles	128	387	(136594896)	*8	8	28	0	8
miles500	Miles	128	1170	8	*20	20	190	0	20
miles750	Miles	128	2113	434	*31	31	465	0	28

La dernière catégorie est composée des instances qui ne font pas partie des deux premières catégories. Le tableau 5.8 présente les résultats pour ces instances. Parmi ces instances sont celles de type *DSJC*. Provenant de graphes (n, p) aléatoires, ces instances sont donc de mauvais candidats à la détection d'IIS et, contrairement à celles de la seconde catégorie, donnent généralement de gros IIS. À l'exception de l'instance *DSJC125.1*, qui a un IIS de 10 variables pour $k = \chi(G)$, la détection d'IIS doit donc se concentrer sur l'obtention de bornes inférieures sur $\chi(G)$ (i.e. on doit utiliser $k \leq \chi(G)$). Les expériences ont ainsi montré, pour l'instance *DSJC125.5*, que $\chi(G) \geq 14$, alors que la meilleure borne connue pour cette instance était $\chi(G) \geq 13$. De plus, les expériences ont montré que $\chi(G) \geq 6$ pour l'instance *DSJC250.1*. Cependant, les bornes inférieures k que l'on peut obtenir pour une instance donnée sont limitées par la taille des IIS trouvés pour cette valeur. Ainsi, alors qu'un IIS a été trouvé pour l'instance *DSJC250.9* avec $k = 50$, cet IIS supposé était trop gros (133 variables) pour que la méthode exacte puisse résoudre le graphe correspondant.

Tableau 5.8 Détection d'IIS de variables sur la troisième catégorie d'instances du challenge DIMACS

Instances						IIS de variables			
nom	type	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	RA	k	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	LB
DSJC125.1	DSJC	125	736	227	*5	10	26	1	4
DSJC125.5	DSJC	125	3891	(71844096)	14	80	1674	37453055	-
DSJC125.9	DSJC	125	6961	(68250955)	40	78	2782	(205206643)	-
DSJC250.1	DSJC	250	3218	(42398413)	6	70	410	3621	-

... *continué sur la page suivante*

Instances						IIS de variables			
nom	type	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	RA	k	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	LB
DSJC250.9	DSJC	250	27897	(33839645)	50	133	8052	(88911252)	-
DSJR500.1	DSJR	500	3555	(141520342)	12	12	66	0	11
DSJR500.5	DSJR	500	58862	(73970922)	26	26	325	0	1
DSJR500.1c	DSJRc	500	121275	(6401403)	63	63	1953	0	2
queen5_5	Queen	25	160	1	*5	5	10	0	5
queen6_6	Queen	36	290	410	*7	25	148	45	7
queen7_7	Queen	49	476	2555	*7	7	21	0	5
queen8_12	Queen	96	1368	(139081460)	*12	12	66	0	11
queen8_8	Queen	64	728	597552	*9	54	538	188021	11
queen9_9	Queen	81	2112	80603809	*10	74	897	135083408	12
queen10_10	Queen	100	2940	(134401345)	10	10	45	0	-
					*11	89	1220	(424776367)	11
queen11_11	Queen	121	3960	(116006580)	*11	11	55	0	6
queen12_12	Queen	144	5192	(101315208)	*12	12	66	0	-
queen13_13	Queen	169	6656	(90800757)	*13	13	78	0	6
queen14_14	Queen	196	8372	(83679129)	*14	14	91	0	-
queen15_15	Queen	225	10360	(69555352)	15	15	105	0	-
queen16_16	Queen	256	12640	(72473005)	16	16	120	0	-
ash331GPIA	HOS	662	4185	14	*4	9	16	2	2
1-FullIns_3	CAR	30	100	7	*4	7	12	1	7
1-FullIns_4	CAR	93	593	5567	*5	15	43	6	14
1-FullIns_5	CAR	282	3247	(106523508)	*6	31	144	271	19
2-FullIns_3	CAR	52	201	1850	*5	9	22	1	9
2-FullIns_4	CAR	212	1621	(209999176)	*6	19	75	8	19
2-FullIns_5	CAR	852	12201	(91922086)	*7	39	244	715	31
3-FullIns_3	CAR	80	346	366830	*5	5	10	0	5
3-FullIns_4	CAR	405	3524	(164058937)	*7	23	116	10	23
3-FullIns_5	CAR	2030	33751	(34366333)	*8	47	371	1675	-
4-FullIns_3	CAR	114	541	80247163	*7	13	51	1	13
4-FullIns_4	CAR	690	6650	(126559559)	*8	27	166	12	25
5-FullIns_3	CAR	154	792	(448858523)	*8	15	70	1	15

Les instances de type *Queen* sont également comprises dans la dernière catégorie. Ces instances sont particulières car les IIS minimum pour $k = \chi(G)$ donnent un CN qui est soit une clique ou qui n'est presque pas réduit. En conséquence, des IIS minimum donnant des cliques ont été trouvés pour les instances *queen5_5*, *queen7_7*, *queen8_12*, *queen11_11*, *queen12_12*, *queen13_13* et *queen14_14*). Il a également été possible de déterminer $\chi(G)$ pour les instances *queen6_6*, *queen8_8* et *queen9_9*

après avoir trouvé des IIS suffisamment petits pour la méthode exacte. Enfin, des bornes inférieure de $k = 10$, $k = 15$ et $k = 16$ ont respectivement été obtenues pour les instances *queen10_10*, *queen15_15* and *queen16_16*.

Le dernier ensemble d'instances de la troisième catégorie est celui de type *CAR* dont le nom contient *FullIns*. Ces instances sont alors des candidats idéaux pour la détection d'IIS. On remarque d'abord que $\chi(G)$ a été obtenu par la méthode exacte pour seulement 5 de ces instances. Par ailleurs, selon les plus récentes publications portant sur le coloriage de graphe, le nombre chromatique des instances *2-FullIns_4*, *2-FullIns_5*, *3-FullIns_4* et *4-FullIns_4* n'a jamais été démontré par une méthode exacte. Pour ces instances, des petits IIS ont été trouvés pour une borne supérieure connue $k \geq \chi(G)$. De plus, comme le nombre chromatique des graphes correspondants, obtenu par la méthode exacte, est une borne inférieure $k \leq \chi(G)$, il a été montré que $\chi(G) = k$. Le nombre chromatique de ces instances a donc été fixé à l'aide de la détection d'IIS.

Pour terminer, les bornes inférieures obtenues par *LB* pour cette catégorie d'instances sont beaucoup plus basses que le nombre de variables des IIS trouvés. Sachant que ces instances ont probablement de gros IIS minimum, on en conclut que *LB* donne de mauvais résultats pour cette catégorie d'instances.

5.9 Temps de calcul des algorithmes de détection

Puisque la procédure *procMWIC* représente la presque totalité de la complexité des algorithmes de détection[8], le temps de calcul de la détection dépend principalement du nombre de fois que cette procédure est appelée (i.e. nombre d'itérations), ainsi que du temps passé à chacun de ces appels. Il a été vu que le nombre d'itérations de

[8] À l'exception de l'algorithme de *hitting set* qui utilise aussi *procMHS*.

détection dépendait de l'algorithme utilisé, de la taille de l'instance, du type d'IIS recherché (i.e contraintes ou variables), de la taille de ces IIS, et du nombre d'erreurs rencontrées. Par ailleurs, le temps passé dans chaque appel à *procMWIC* dépend surtout de la difficulté à résoudre le MWCSP ou le MPWCSP correspondant. Ainsi, si la méta-heuristique est arrêtée trop tôt, l'affectation retournée est possiblement sous-optimale, et des erreurs surviendront. Ces erreurs peuvent causer l'échec de la détection. Par contre, si on laisse la méta-heuristique s'exécuter pendant longtemps (i.e. grand nombre d'itérations de recherche locale et de relances), l'affectation retournée aura plus de chances d'être optimale, mais la détection sera également plus longue. Voici quelques indications sur le temps de calcul total requis pour la détection d'IIS lors de l'expérimentation.

Considérons la détection d'IIS pour les instances aléatoires, dont les résultats sont donnés dans les tableaux I.1 et I.2. Le jeu de paramètres utilisé pour *procMWIC* dépend de la difficulté de l'instance (*voir tableau 5.1*). Pour $p = 0.1$ et $n \in \{100, 110\}$, le jeu utilisé est celui des instances faciles, et la détection, à l'aide de l'algorithme d'insertion, a pris moins de 5 minutes[9]. Par ailleurs, pour $n \in \{120, 130, 140\}$, le jeu de paramètres utilisé est celui des instances moyennes, et la détection a pris moins de 30 minutes. Par contre, pour $n \geq 150$, le jeu de paramètres employé est celui des instances difficiles. La durée de la détection, pour ces instances, a donc considérablement augmenté. De plus, la durée de la détection a également augmenté avec la taille des instances. Ainsi, pour $n = 150$ la détection a mis moins de 2 heures, alors que pour $n = 220$, la détection a pris jusqu'à 10 heures. Par ailleurs, pour toutes les instances avec $p = 0.5$, le jeu de paramètres utilisé est celui des instances difficiles. La détection a donc mis, dans les pires cas, plusieurs heures. Cependant, l'algorithme de pré-filtrage a permis de réduire la durée de la détection. L'étape de pré-filtrage prend de quelques secondes, pour

[9]Environ le double du temps est requis pour l'algorithme de retrait.

les instances faciles, à quelques minutes pour les instances difficiles. Par contre, suite à cette étape, la durée de la détection, faite avec l'algorithme de retrait ou d'insertion, a été, dans certains cas, réduite de moitié.

On note cependant que la détection a été moins longue pour les instances qui ne sont pas aléatoires. Ainsi, en utilisant la technique d'accélération de la section 4.4, la détection a pris moins de 10 minutes, pour toutes les instances du challenge DIMACS provenant de graphes sommet-critiques. Ensuite, pour les instances contenant un IIS minimum, correspondant aux sommets d'une clique, l'algorithme de pré-filtrage a généralement isolé cet IIS en quelques minutes, de telle sorte que la détection a mis au total moins d'une heure pour ces instances. Enfin, le temps requis pour obtenir une borne inférieure par l'algorithme de *hitting set* ou l'algorithme de la section 3.3.5 dépend de la qualité recherchée de ces bornes. Ainsi, dans le cas de l'algorithme de *hitting set*, une heure est généralement suffisante pour converger vers une borne (i.e. les itérations suivantes sont, à toute fin pratique, inutiles), alors que quelques minutes seulement sont nécessaires pour le second algorithme.

CHAPITRE 6

CONCLUSION

Dans ce mémoire ont été présentés des algorithmes pour trouver des IIS de contraintes et de variables dans un CSP incohérent. Ainsi, les algorithmes de retrait, d'insertion et de *hitting set* garantissent l'obtention d'IIS en un nombre fini d'itérations. Par ailleurs, il a été montré que l'algorithme de *hitting set* pouvait obtenir des IIS de cardinalité minimum, si la procédure *procMHS* retournait des HS de cardinalité minimum. De plus, il a été vu que cet algorithme prenait un nombre exponentiel d'itérations pour trouver un tel IIS. Cependant, il est possible d'obtenir, en tout temps, une borne inférieure[1] sur la taille d'IIS minimum avec la taille du dernier HS retourné par *procMHS*. Une borne inférieure sur la taille d'IIS minimum peut également être obtenue à l'aide de l'algorithme décrit à la section 3.3.5.

Des techniques complémentaires de détection ont également été proposées. Parmi ces techniques, l'algorithme hybride est une variante de l'algorithme de retrait qui utilise les poids des contraintes ou des variables pour améliorer, dans certains cas, la détection. Il a été montré que cet algorithme garantissait également la détection d'un IIS dans un nombre fini d'itérations. De plus, l'algorithme de retour-arrière, basé sur l'algorithme d'insertion, explore un arbre de recherche où les noeuds sont les contraintes violées ou les variables désinstanciées à chaque itération, et les branches la contrainte ou la variable de cet ensemble qui est conservée. Il a été montré que cet algorithme permettait, en un nombre exponentiel d'itérations,

[1] Cette borne est exacte seulement si *procMHS* retourne des HS de cardinalité minimale, sinon il s'agît d'une borne sur la taille d'un IIS quelconque.

d'obtenir le plus petit IIS pouvant être obtenu par l'algorithme d'insertion. Des heuristiques permettant d'obtenir de plus petits IIS ont également été décrites. Ainsi, l'algorithme de pré-filtrage peut être utilisé comme pré-traitement pour filtrer le plus possible de contraintes ou de variables, dans le but d'isoler un IIS de petite taille, ensuite trouvé par un algorithme de détection. Enfin, l'heuristique de poids de voisinage utilise le poids des contraintes ou des variables pour guider les algorithmes de détection vers l'obtention d'un petit IIS.

Il a été vu que les procédures *procMWIC* et *procMHS* utilisées par ces algorithmes devaient respectivement résoudre le MWCSP ainsi que le MPWCSP, et le MHS, qui sont trois problèmes NP-difficiles. La détection d'IIS, utilisant des algorithmes exacts pour *procMWIC* et *procMHS*, est alors impraticable. Des méta-heuristiques pour résoudre le MWCSP et le MPWCSP, basées sur la recherche locale Tabou, ont alors été décrites[2]. Ainsi, une version heuristique des algorithmes de retrait, d'insertion et de *hitting set*, utilisant ces méta-heuristiques, a été proposée. Cependant, il a été montré que les affectations sous-optimales retournées par ces méta-heuristiques causaient des erreurs pouvant mener à l'obtention d'un ensemble cohérent de contraintes ou de variables. Pour déceler et corriger ces erreurs, des stratégies de récupération d'erreur ont alors été ajoutées à ces algorithmes de détection.

Des expériences ont par ailleurs été menées sur des instances générées aléatoirement ou connues du problème de coloriage de graphes. Ces expériences avaient pour but d'évaluer les principaux algorithmes et techniques de détection, ainsi que de montrer l'utilité des IIS pour prouver qu'un CSP est incohérent. Dans un premier temps, la détection d'IIS de contraintes a été comparée avec la détection d'IIS de variables. Cette expérience a d'abord montré que certains algorithmes et heuris-

[2]L'implantation de *procMHS* a été brièvement abordée dans la section 5.1

tiques de détection étaient plus efficaces lors de la détection d'IIS de contraintes que des IIS de variables, alors que d'autres donnaient de meilleurs résultats pour la détection d'IIS de variables. De plus, il a été constaté que la détection d'IIS de variables était beaucoup plus rapide, et que les IIS de variables donnaient des problèmes réduits plus faciles à résoudre que ceux obtenus avec les IIS de contraintes. La deuxième expérience, qui portait sur l'algorithme de *hitting set* a permis d'observer, pour les instances testées, que le nombre d'itérations mis par cet algorithme pour obtenir un IIS de variables est exponentiel en la taille de l'instance. Ensuite, la troisième expérience, qui comparait les heuristiques de détection, a permis de constater que ces heuristiques permettent aux algorithmes de retrait et d'insertion d'obtenir, de façon constante, des IIS de variables plus petits. Par ailleurs, l'expérience a également déterminé que l'utilisation de l'algorithme de pré-filtrage donnait de plus petits IIS que l'heuristique de poids de voisinage. L'expérience suivante, qui comparait les bornes inférieures sur la taille d'IIS minimum obtenues par l'algorithme de *hitting set* et l'algorithme de la section 3.3.5, a montré que le premier algorithme donne de meilleures bornes pour des instances de petite ou moyenne taille. Les deux dernières expériences avaient pour but d'utiliser la détection d'IIS de variables pour déterminer le nombre chromatique de graphes. La première de ces deux expériences était menée sur des instances provenant de graphes générés aléatoirement. Cette expérience a permis de constater que la réduction du nombre de variables de l'instance diminue lorsque la densité de contrainte p et le nombre chromatique k de l'instance augmentent. Par contre, pour des valeurs données de p et k, cette réduction augmente lorsque la taille de l'instance augmente. Il a ainsi été possible d'obtenir le nombre chromatique de graphes réduits, alors que cette valeur n'avait pas été obtenue pour les graphes originaux. La dernière expérience, qui portait sur des instances du challenge DIMACS, a montré que la technique d'accélération de la section 4.4 pouvait identifier les instances dont les variables forment un IIS en quelques itérations. Finalement,

les résultats de cette expérience ont permis d'améliorer la borne inférieure sur le nombre chromatique d'instances connues, et même de fixer le nombre chromatique de quatre instances pour lesquelles cette valeur n'était pas connue.

Pour terminer, plusieurs améliorations pourraient être apportées aux algorithmes de détection présentés dans ce mémoire. Ainsi, il serait bien de rendre ces algorithmes plus robustes, sans augmenter la durée de la détection. De plus, il serait intéressant d'avoir, pour l'algorithme de *hitting set*, une heuristique permettant d'obtenir des IIS plus rapidement. Enfin, on note que ces algorithmes pourraient être testés sur des CSP modélisant d'autres problèmes, comme par exemple, le problème SAT.

RÉFÉRENCES

AMALDI, E., PFETSCH, M.E., TROTTER, L.E., "On the Maximum Feasible Subsystem Problem, IISs, and IIS-hypergraphs", *Math. Program.*, vol.95/3, pp.533-554, 2003

AMILHASTRE, J., FRAGIER, H., MARQUIS, P., "Consistency Restoration and Explanations in Dynamic CSPs - Application to Configuration", *Artificial Intelligence*, vol.135/2002, pp.199-234, 2002

ARAGON, C.R., JOHNSON, D.S., MCGEOCH, L.A., SCHEVON, C., "Optimization by Simulated Annealing: an Experimental Evaluation. Part II, Graph Coloring and Number Partitioning", *Operations Research*, vol.39, pp.378-406, 1991

BRÉLAZ, D., "New Methods to Color the Vertices of a Graph", *Communications of the ACM*, vol.22/4, pp.251-256, 1979

BROWN, J.R., "Chromatic Scheduling and the Chromatic Number Problem", *Management Science*, vol.19/4, pp.456-463, 1972

CARVER, W.B., "Systems of Linear Inequalities", *Annals of Mathematics*, vol.23, pp.212-220, 1921

CHAKRAVARTI, N., "Some Results Concerning Post-infeasibility Analysis", *European Journal of Operational Research*, vol.73, pp.139-143, 1994

CHINNECK, J.W., "Finding a Useful Subset of Constraints for Analysis in an Infeasible Linear Program", *INFORMS Journal on Computing*, vol.9/2, 1997

CHINNECK, J.W.. "Feasibility and Viability", Advances in Sensitivity Analysis and Parametric Programming, T. Gal and H.J. Greenberg (eds.), Kluwer Academic Publishers, *International Series in Operations Research and Management Science*, vol.6, 1997

CHOUEIRY, B.Y., "Abstraction Methods for Resource Allocation", Thèse de doctorat, École Polytechnique Fédérale de Lausanne, 1994

COOK, S., "The Complexity of Theorem Proving Procedures", *Proc. 3rd Ann. ACM Symp on Theory of Computing*, pp.151-158, 1971

DAVIS, M., PUTNAM, H., "A Computing Procedure for Quantification Theory", *Journ. Of the ACM*, vol.7, pp.201-215, 1960

FLEURENT, C., FERLAND, J.A., "Genetic and Hybrid Algorithms for Graph Coloring", *Annals of Operations Research*, vol.63, pp.437-461, 1996

FOX, M.S., SADEH-KONIECPOL, N., "Why is Scheduling Difficult? A CSP Perspective", *Proc. of the Ninth European Conference on Artificial Intelligence address: Stockholm Sweden month: August*, pp.754-767, 1990

FREUDER, E.C., WALLACE, R.J., "Partial Constraint Satisfaction", *Artificial Intelligence*, vol.58/1, pp.21-70, 1992

GALINIER, P., HAO, J.K., "Tabu Search for Maximal Constraint Satisfaction Problems", Proc. of the Third International Conference Principles and Practice of Constraint Programming, *Lecture Notes in Computer Science*, vol.1330, pp.196-208, Springer Verlag, Berlin, 1997

GALINIER, P., HAO, J.K., "Hybrid evolutionary algorithms for graph coloring". *Journal of Combinatorial Optimization*, vol.3/4, pp.379-397, 1999

GALINIER, P., HERTZ, A., "Solution Techniques for the Large Set Covering Problem", Rapport technique G-2003-44, *Les Cahiers du GERAD*, Montréal, Canada, 2003

GAREY, M.R., JOHNSON, D.S., "Computers and Intractability : A Guide to the Thoery of NP-Completeness", *W.H. Freman and Company*, New York, 1979

DE GIVRY, S., LARROSA, J., MESEGUER, P., SCHIEX, T., "Solving Max-SAT as Weighted CSP", *Proc. of CP'2003*, Cork, Irlande, Octobre, 2003

GLESSON, J., RYAN, J., "Identifying Minimally Infeasible Subsystems of Inequalities", *ORSA Journal on Computing*, vol.2, pp.61-63, 1990

GREENBERG, H.J., "An empirical Analysis of Infeasibility Diagnosis for Instances of Linear Programming Blending Models", *IMS Journal of Mathematics in Business & Industry*, vol.4, pp.163-210, 1992

HERRMANN, F., HERTZ, A., "Finding the Chromatic Number by Means of Critical Graphs", *ACM Journal of Experimental Algorithmics*, vol.7/10, pp.1-9, 2002

JOHNSON, D.S., TRICK, M.A., 1996, "Proceedings of the 2nd DIMACS Implementation Challenge", DIMACS Series in Discrete Mathematics and Theoretical Computer Science, *American Mathematical Society*, vol.26, 1996

KNUTH, D.E., "The Stanford Graphbase: A Platform for Combinatorial Computing", *ACM Press*, 1993

GELATT, C.D., KIRKPATRICK, S., VECCHI, M.P., "Optimization by Simulated Annealing", *Science*, vol.220, pp.671-680, 1983

GLOVER, F., "Tabu Search - Part I", *ORSA Journal on Computing*, vol.1/3, pp.190-206, 1989

GLOVER, F., "Tabu Search - Part II", *ORSA Journal on Computing*, vol.2/1, pp.4-32, 1989

KONDRAK, G., "A Theoretical Evaluation of Selected Backtracking Algorithms", *Rapport technique TR-94-10*, Département de Sciences Informatiques, Université de l'Alberta, Edmonton, Alberta, Canada, 1994

KUBALE, M., JACKOWSKI, B., "A Generalized Implicit Enumeration Algorithm for Graph Coloring", *Communications of the ACM* vol.28/4, pp.412-418, 1985

LAROSSA, J., MESEGUER, P., "Optimisation-based Heuristics for Maximal Constraint Satisfaction", *Proc. Of CP-95*, pp.190-194, Cassis, France, 1995

MACKWORTH, A.K., "Constraint Satisfaction", *S.C. Shapiro (Ed.) Encyclopedia on Artificial Intelligence*, John Wiley & Sons, NY, 1987

MAZURE, B., SAÏS, L., GRÉGOIRE, E., "Boosting complete techniques thanks to local search methods", *Annals of Mathematics and Artificial Intelligence*, vol.22, pp.319-331, 1998

MEHROTRA, A., TRICK, M.A., "A Column Generation Approach for Exact Graph Coloring", *INFORMS Journal on Computing*, vol.8, pp.344-354, 1996

MITTAL, S., FALKENHAINER, B., "Dynamic Constraint Satisfaction Problems", *Proc. of AAAI-90*, pp.25-32, Boston, Massachusetts, 1990

MORGENSTERN, C., "Distributed Coloration Neighborhood Search", D.S. Johnson and M.A. Trick, eds. Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, *American Mathematical Society*, vol.26, pp.335-357, 1996

PAPADIMITRIOU, C.H., STEIGLITZ, K., "Combinatorial Optimization", Englewood Cliffs, *Prentice-Hall*, 1982

PARKER, M.R., "A Set Covering Approach to Infeasibilty Analysis of Linear Programming Problems and Related Issues", Thèse de doctorat, Département de Mathématiques, Université du Colorado, Denver, Colorado, 1995

PARKER, M.R., RYAN, J., "Finding the Maximum Weight Feasible Subsystem of an Infeasible System of Inequalities", *Annals of Mathematics and Artificial Intelligence*, vol.17, pp.107-126, 1996

PEEMÖLLER, J., "A Correction to Brélaz's Modification of Brown's Coloring Algorithm", *Communications of the ACM*, vol.26/8, pp.593-597, 1983

SELMAN, B., LEVESQUE, H., MITCHELL, D., "A New Method for Solving Hard Satisfiability Problems", *Proc. AAAI-92*, pp.440-446, 1992

SMITH, I., FALTINGS, B., "Implementing Qualitative Reasoning for Structural Design Using Constraint Propagation", *Computing in Civil Engineering*, pp.1251-1258, 1993

STUMPTNER, M., "An Overview of Knowledge-based Configuration", *AI Communications*, pp.111-125, 1997

TAMIZ, M., MARDLE, S., JONES, D., "Detecting IIS in Infeasible Linear Programs Using Techniques from Goal Programming", *Computers and Operations Research*, vol.23/2, pp.113-119, 1996

TSANG, E., "Foundations of Constraint Satisfaction", *Academic Press*, London, 1993

VAN LOON, J., "Irreducibily Inconsistent System of linear inequalities", *European Journal of Operational Research*, vol.8, pp.283-288, 1981

WALLACE, R.J., "Enhancements of Branch-and-bound Methods for the Maximal Constraint Satisfaction Problem", *Proc of AAAI-96*, pp.188-196, Portland, Oregon, 1996

WALLACE, R.J., "Analysis of Heuristics Methods for Patial Constraint Satisfaction Problems", *Proc. Of CP-96*, vol.1118, pp.308-322, Cambridge, Massachusetts, 1996

ANNEXE I

TABLEAUX DE RÉSULTATS

Instances				IIS de variables								
				Ret+h			Ins+h			FI		
$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	k	RA	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$
100	496	5	92	38	137	109	55	207	271	38	138	122
100	447	5	193	66	263	233	62	234	333	63	242	282
100	499	5	33	38	137	63	44	160	66	42	156	55
100	507	5	135	46	175	105	51	198	151	42	151	237
110	600	5	114	34	119	32	61	244	201	41	151	116
110	555	5	45	36	126	40	50	188	98	36	123	97
110	592	5	131	33	116	58	46	170	330	33	113	30
110	610	5	72	43	162	117	51	188	57	43	157	76
120	715	5	7105	32	113	59	52	199	157	42	156	31
120	669	5	45	21	63	10	38	134	16	33	108	32
120	714	5	1172	31	109	67	50	195	38	36	126	121
120	706	5	186	27	88	71	61	239	693	43	157	102
130	795	5	173090	31	108	30	42	150	87	35	115	39
130	832	5	62743	34	120	47	46	171	59	42	154	41
130	828	6	1519301	113	712	1493884	112	702	1898057	113	710	2094563
130	843	6	462073	100	617	415567	101	617	519210	99	605	323833
140	972	6	767916	99	625	1551179	102	637	1122434	99	615	970417
140	936	6	1130903	103	640	1610227	107	664	1439572	104	647	2775637
140	988	6	138308	83	499	137685	94	569	202552	88	522	149205
140	968	6	906265	105	654	4436370	108	681	1621338	104	643	3149618
150	1103	6	829224	96	610	1289371	99	622	887587	97	606	1164246
150	1098	6	109821	93	579	247208	95	581	148611	93	565	202147
150	1120	6	79497	81	481	133880	87	517	234917	88	531	274521
150	1128	6	194667	91	554	115757	98	606	112071	98	591	298509
160	1261	6	187661	87	539	347890	103	642	407695	94	582	520562
160	1251	6	101953	83	501	140258	95	580	126134	92	561	96891
160	1274	6	107938	79	474	71615	91	536	148744	83	497	81926
160	1279	6	199720	87	529	250846	103	647	7955344	93	564	257126
170	1430	6	(43603)	80	487	168991	97	597	340743	89	550	167544

... continué sur la page suivante

Instances				IIS de variables								
				Ret+h			Ins+h			FI		
$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	k	RA	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$
170	1414	6	8320828	80	477	85597	96	581	490732	85	500	94995
170	1460	6	(45118)	69	405	7660	80	470	7367	73	421	10720
170	1440	6	70336660	85	516	193161	96	581	61871	93	563	207389
180	1603	6	(46199)	78	467	10347	102	613	294620	87	519	44426
180	1617	7	(44611)	158	1386	(41491)	159	1386	(42298)	158	1389	(41474)
180	1584	7	(45690)	168	1471	(45480)	168	1467	(45627)	168	1467	(45627)
180	1639	7	(43834)	148	1278	(37592)	148	1266	(39431)	150	1286	(39265)
190	1799	7	(49624)	147	1281	(39683)	156	1359	(41245)	152	1327	(41473)
190	1784	7	(47708)	152	1327	(41621)	157	1352	(43114)	154	1337	(41929)
190	1826	7	(49295)	140	1206	(38256)	152	1307	(40572)	141	1206	(38298)
190	1785	7	(49653)	158	1389	(44149)	160	1403	(43280)	160	1405	(44872)
200	2036	7	(50830)	139	1220	(46450)	147	1291	(46855)	148	1281	(45411)
200	1941	7	(48910)	153	1343	(63880)	155	1347	(64474)	155	1347	(63452)
200	2026	7	(46221)	130	1113	(44259)	150	1288	(60526)	148	1277	(47730)
200	1977	7	(47076)	146	1278	(46846)	147	1281	(47080)	152	1324	(48111)
210	2221	7	(61537)	134	1166	(34091)	148	1289	(37564)	144	1256	(37077)
210	2123	7	(54287)	146	1277	(38301)	157	1367	(39882)	154	1342	(40706)
210	2226	7	(64066)	129	1109	(33831)	143	1213	(35316)	136	1161	(35703)
210	2175	7	(59757)	139	1210	(36732)	153	1331	(38900)	149	1283	(38497)
220	2422	7	(62081)	132	1145	(32986)	148	1279	(37429)	139	1208	(35358)
220	2348	7	(59384)	137	1188	(34917)	149	1290	(37488)	148	1281	(36713)
220	2435	7	(56396)	125	1070	(31717)	146	1259	(37417)	134	1144	(34620)
220	2387	7	(58686)	135	1166	(34208)	145	1241	(34873)	144	1245	(35136)

Tableau I.1 Détection d'IIS de variables sur des instance aléatoires avec $p = 0.1$

Instances				IIS de variables								
				Ret+h			Ins+h			FI		
$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	k	RA	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$
80	1547	12	14029599	64	1051	1285350	66	1086	3887945	63	1007	726965
80	1528	12	12860059	59	911	293100	60	926	531284	58	884	429381
80	1609	13	7255750	75	1440	21670437	73	1380	6590146	72	1343	6387828
80	1582	13	58684771	75	1425	22694200	74	1389	14340999	73	1350	20529374
90	1924	13	128822599	74	1398	17898027	75	1402	30015939	71	1288	17201820
90	1984	13	133509732	72	1354	4320820	73	1374	7693584	72	1334	11737502
90	1978	14	(26310)	88	1908	(26010)	87	1870	(26268)	87	1870	(26253)
90	2003	14	(26271)	86	1858	(25914)	84	1783	(25912)	84	1783	(26381)
95	2223	14	(28823)	82	1723	(27079)	83	1754	(27116)	82	1703	(26512)
95	2149	14	(27836)	89	1931	(26961)	89	1935	(26994)	88	1896	(27510)

... continué sur la page suivante

Instances				IIS de variables								
				Ret+h			Ins+h			FI		
$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	k	RA	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$	$\|\mathcal{X}'\|$	$\|\mathcal{C}'\|$	RA$'$
95	2223	14	(28861)	83	1777	(27784)	88	1906	(26981)	84	1797	(27024)
95	2208	14	(28553)	85	1834	(26831)	84	1780	(34611)	86	1841	(26097)
100	2381	14	(30027)	83	1748	(25375)	87	1869	(25959)	83	1724	(26025)
100	2444	14	(30107)	81	1690	(26861)	85	1828	(25663)	82	1711	(25304)
100	2469	15	(30328)	97	2345	(28932)	96	2297	(28595)	97	2342	(28454)
100	2465	15	(31121)	100	2465	(29744)	99	2425	(29499)	99	2414	(29335)

Tableau I.2 Détection d'IIS de variables sur des instance aléatoires avec $p = 0.5$